

UNIVERSIDAD AUTONOMA DE MADRID
ESCUELA POLITECNICA SUPERIOR



Grado en Ingeniería Informática

TRABAJO FIN DE GRADO

**Modelado de gramáticas basadas en dependencias a partir de
un treebank de constituyentes**

Rebeca de la Paz González
Tutor: Pablo Alfonso Haya Coll

Julio 2017

Modelado de gramáticas basadas en dependencias a partir de un treebank de constituyentes

AUTOR: Rebeca de la Paz González
TUTOR: Pablo Alfonso Haya Coll

Dpto. Ingeniería Informática
Escuela Politécnica Superior
Universidad Autónoma de Madrid
Julio de 2017

Resumen (castellano)

Este Trabajo Fin de Grado tiene como objetivo iniciar la transformación del modelo gramatical utilizado actualmente, conocido como modelo de constituyentes, a un modelo de dependencias. La transformación llevada a cabo se realiza sobre un conjunto de oraciones en formato de constituyentes, denominado treebank, el cual ha sido desarrollado por el departamento de lingüística de la Universidad Autónoma de Madrid, a lo largo de muchos años.

En el documento se explica a qué se debe la transformación de constituyentes a dependencias, haciendo referencia a proyectos reales llevados a cabo por importantes instituciones como son Google o la Universidad de Stanford en Estados Unidos, y de los cuales se ha realizado un estudio previo para poder llegar a obtener un resultado lo más fiable posible, acorde a lo que existe actualmente.

La finalidad de este proyecto es desarrollar un algoritmo, en lenguaje Java, que permita llevar a cabo las transformaciones necesarias para generar el nuevo treebank de dependencias.

Para poder llegar a construir el treebank de dependencias deseado, ha sido necesario llevar a estudio los trabajos realizados por la Universidad de Stanford y el proyecto de Universal Dependencies. Para este estudio se ha necesitado la ayuda de los lingüistas de la Universidad Autónoma debido a que, antes de poder desarrollar el código, era necesario tomar decisiones de importancia sobre el tratamiento de algunos elementos y se necesitaba entender completamente como se realiza la transformación de forma teórica.

Debido a limitaciones de carácter lingüístico que se han encontrado durante el desarrollo de este proyecto, no todos los elementos de constituyentes tienen una equivalencia correcta a dependencias. Estas limitaciones se deben a que es difícil designar una regla genérica a la transformación, puesto que existen diferentes formas de transformación y dependiendo del contexto estas pueden ser válidas o no. Teniendo esto en cuenta, es necesario realizar una profunda reflexión para tomar la mejor decisión.

Palabras clave (castellano)

Treebank, dependencias, constituyentes, analizador, root

Abstract (English)

The current bachelor's thesis aims to initiate the transformation of the currently used grammatical model, known as constituent grammar model, to a dependency grammar model. The applied transformation is carried out on a set of constituent format sentences, called treebank, which has been developed by the linguistics department of Universidad Autónoma de Madrid over many years.

The document explains the reasons for the transformation of constituents to dependencies, referring to real projects carried out by important institutions such as Google or Stanford University in the United States. A previous research has been made to these projects in order to obtain a result as reliable as possible, according to what currently exists.

The purpose of this project is to develop an algorithm using the Java programming language. The algorithm will carry out the proper transformations to obtain a new dependency treebank.

In order to be able to build the desired dependency treebank, it has been necessary to analyse the works realised by the Stanford University and the Universal Dependencies project related to this subject. To conduct the analysis the help of the linguists of Universidad Autónoma has been required because, before being able to develop the code, it was needed to take decisions on important matters about how to treat some elements and fully understand how to realize the transformations theoretically.

Due to linguistic limitations that have been encountered during the development of this project, not all elements of constituents have a correct equivalence to dependencies. These limitations are due to the fact that it is difficult to designate a generic rule for transformation since there are different forms of transformation and depending on the context they may be valid or not. With this in mind, it is necessary to make a deep reflection to make the best decision.

Keywords (inglés)

Treebank, dependencies, constituencies, parser, root

Agradecimientos

Quiero agradecer a mi familia, en particular a mis padres, y amigos, aquellos que conocí en la universidad, por todo el apoyo y cariño que he recibido por su parte, y que tanto bien me ha hecho a lo largo de estos años.

Agradecer a mi tutor, Pablo Haya, por toda su ayuda a lo largo del desarrollo del trabajo, sin olvidarme también de Antonio Moreno Sandoval, del departamento de lingüística de la universidad, que tanto me ha ayudado en este área.

Gracias a todos, de corazón.

INDICE DE CONTENIDOS

1	Introducción	1
1.1	Motivación	1
1.2	Objetivos	1
1.3	Organización de la memoria	2
2	Estado del arte	3
2.1	Análisis sintáctico	3
2.1.1	Constituyentes	4
2.1.2	Dependencias	6
2.2	Stanford Dependencies	7
2.3	Google Dependencies	8
2.4	Universal Dependencies	9
2.5	Formato de representación	10
2.5.1	Formato Stanford	10
2.5.2	Formato CoNLL	11
3	Diseño	13
3.1	Análisis	13
3.1.1	Oraciones y constituyentes	14
3.1.2	Transformación de árboles de constituyentes	16
3.2	Limitaciones	19
4	Desarrollo	21
4.1	Plantilla en formato CoNLL	21
4.2	Formato de ficheros	22
4.2.1	Formato de entrada	22
4.2.2	Formato de salida	23
4.3	Implementación	23
5	Pruebas	31
5.1	Pruebas de caja negra	31
5.2	Pruebas de integración	31
6	Calidad de software	33
7	Conclusiones y trabajo futuro	35
7.1	Conclusiones	35
7.2	Trabajo futuro	35
	Referencias	37
	Glosario	39
	Anexos	I
A	Manual de instalación	I
B	Manual del programador	II
C	Cobertura de las pruebas	III

INDICE DE FIGURAS

FIGURA 1. EJEMPLO DE LA PRIMERA DECLINACIÓN DEL RUSO	4
FIGURA 2. EJEMPLO DE ÁRBOL DE CONSTITUYENTES EN INGLÉS MEDIANTE LA HERRAMIENTA DE STANFORD.	5
FIGURA 3. PRIMER EJEMPLO DE ÁRBOL DE CONSTITUYENTES EN ESPAÑOL.	5
FIGURA 4. SEGUNDO EJEMPLO DE ÁRBOL DE CONSTITUYENTES EN ESPAÑOL.....	6
FIGURA 5. ÁRBOL DE DEPENDENCIAS EN ESPAÑOL	6
FIGURA 6. ÁRBOL DE DEPENDENCIAS DE STANFORD	7
FIGURA 7. ÁRBOLES DE DEPENDENCIA DE GOOGLE	9
FIGURA 8. REPRESENTACIÓN DEL EJEMPLO DE <i>STANFORD DEPENDENCIES</i> GENERADA CON GRAPHVIZ	11
FIGURA 9. REPRESENTACIÓN DE LAS RELACIONES DE DEPENDENCIAS CON FORMATO CONLL DE <i>UNIVERSAL DEPENDENCIES</i>	12
FIGURA 10. REPRESENTACIÓN DE GRAFO DEL EJEMPLO DE <i>UNIVERSAL DEPENDENCIES</i> GENERADA CON GRAPHVIZ.....	12
FIGURA 11. ÁRBOL DE CONSTITUYENTES DE EJEMPLO.....	13
FIGURA 12. REPRESENTACIÓN DEL ÁRBOL DE CONSTITUYENTES DE EJEMPLO	13
FIGURA 13. SINTAGMA PREPOSICIONAL	14
FIGURA 14. SINTAGMA NOMINAL	14
FIGURA 15. ÁRBOL DE CONSTITUYENTES CON SUJETO ELIDIDO.	16
FIGURA 16. SUBÁRBOL QUE SE USARÁ PARA REFLEJAR EL SEGUIMIENTO DEL ALGORITMO.....	17
FIGURA 17. ÁRBOL CON PROBLEMA DE PRONOMBRE	20
FIGURA 18. PLANTILLA EN FORMATO CONLL EN ESPAÑOL.....	21
FIGURA 19. FICHERO DE ORACIONES ORIGINALES EN TEXTO PLANO	22
FIGURA 20. FICHERO DE ORACIONES COMO ÁRBOLES DE CONSTITUYENTES	22
FIGURA 21. FICHERO DE RELACIONES.....	23
FIGURA 22. FORMATO CONLL TRAS APLICAR EL ALGORITMO	23
FIGURA 23. CLASE PRINCIPAL DE EJECUCIÓN DEL ALGORITMO	24

FIGURA 24. DIAGRAMA DE CLASES DE LA LECTURA DEL FICHERO DE ETIQUETAS	25
FIGURA 25. REPRESENTACIÓN MEDIANTE CORCHETES DE UN ÁRBOL DE CONSTITUYENTES	26
FIGURA 26. CLASE RELATION	26
FIGURA 27. DIAGRAMA DE CLASES ENFOCADO EN LA CLASE CONSTToDEPEND	27
FIGURA 28. CÓDIGO DEL MÉTODO RECURSIVEFUNCTION(NODE): ARRAYLIST<RELATION>	28
FIGURA 29. CÓDIGO DEL MÉTODO CREATERELATION(NODE): RELATION	29
FIGURA 30. CÓDIGO DEL MÉTODO COMPLETERELATION(ARRAYLIST<RELATION>): VOID	30

INDICE DE TABLAS

TABLA 1. TABLA RESUMEN DE CLÁUSULAS DE CONSTITUYENTES	15
TABLA 2. TABLA RESUMEN DE LAS PRUEBAS DE CAJA NEGRA	31
TABLA 3. TABLA RESUMEN DEL ANÁLISIS DE LA HERRAMIENTA KIUWAN	34
TABLA 4. COBERTURA DE LA CLASE READTEST	III
TABLA 5. COBERTURA DE LA CLASE RELATIONTEST	IV
TABLA 6. COBERTURA DE LA CLASE TUPLETEST	IV
TABLA 7. COBERTURA DE LA CLASE FIELDTEST	V
TABLA 8. COBERTURA DE LA CLASE FIELDRELATIONTEST	V

1 Introducción

1.1 Motivación

Este trabajo surge debido a la problemática que presenta en algunas ocasiones el modelo de gramática de constituyentes que es la representación predominante de lo que conocemos como análisis sintáctico.

El análisis sintáctico se puede considerar la parte central del análisis lingüístico, que necesita del análisis morfológico para poder conocer la categoría sintáctica y las propiedades gramaticales de las unidades que forman las oraciones. Por tanto, se puede concluir que la sintaxis es la rama de la lingüística que estudia las reglas, principios y normas que rigen la combinatoria de las palabras o secuencias de estas que forman sintagmas y oraciones.

El modelo predominante hasta ahora ha sido la gramática de constituyentes desde que lo introdujo Chomsky en la década de los setenta pues el modelo descriptivo que se utilizaba hasta entonces y que únicamente aplicaba el método de los constituyentes inmediatos, por lo que muchas estructuras de oraciones quedaban fuera. A partir de ellos surge el modelo de constituyentes que se mantiene hasta ahora como elección principal ya que su aplicación principal al inglés ha dado buenos resultados.

Sin embargo, nuevos estudios revelan que no es tan eficaz para otras lenguas con una estructura tan definida y fija como el inglés. Este idioma siempre mantiene dentro de unos límites el formato típico de una oración y que se enseña en el colegio, es decir, sujeto y predicado, con sus particularidades en cada una de las partes, pues pueden contener otras composiciones dentro de ellas.

Esa estructura que prácticamente es fija para el inglés no siempre se cumple para otros idiomas, como por ejemplo en el nuestro, el español. En el castellano existen oraciones que mantienen ese formato, pero hay otras en las que los elementos se encuentran invertidos, intercalados o puede que no lleguen a existir.

El elemento principal con el que se trabaja es un treebank, que se puede describir como un archivo que recopila numerosas frases donde cada oración es representada con una misma estructura, normalmente árboles, y que es previamente anotada ajustándose a la gramática formal empleada. Los treebanks siempre están en continua evolución, ya que se van añadiendo nuevos ejemplos que se van teniendo en cuenta o bien para la posible corrección de fallos.

Con los problemas comentados anteriormente se plantea la transformación del treebank de constituyentes, desarrollado por el departamento de lingüística de la Universidad Autónoma de Madrid, a uno de dependencias, en el cual las relaciones existen entre términos y no entre subestructuras dentro de la oración.

La finalidad futura cuando se complete todo el treebank es poder realizar entrenamientos mediante modelos de aprendizaje automático para algunos de los analizadores sintácticos existentes, de esta forma se podría comprobar que como fallos que ocurrían con los modelos de constituyentes se solventan con las dependencias, haciendo que el aprendizaje para lenguas con rica morfología.

1.2 Objetivos

El objetivo de este proyecto consiste en crear un algoritmo capaz de realizar una transformación de un treebank de constituyentes del español a uno de dependencias para la misma lengua.

El banco de oraciones en formato de constituyentes, es decir, en representación de árboles binarios, que se va a usar, ha sido cuidadosamente creado por el departamento de lingüística de la UAM a lo largo de varios años.

Para realizar la transformación entre los modelos, se han seleccionado dos representaciones diferentes para el mostrar la treebank de dependencias, una usada por la Universidad de Stanford¹, la cual se ha escogido por su sencillez a la hora de mostrar las relaciones entre elementos, y otra que pese a ser un poco más compleja, es la que se pretende usar de manera universal por la plataforma Universal Dependencies².

La transformación entre las diferentes representaciones nombradas anteriormente se ha realizado diseñando e implementando un algoritmo en Java, que partiendo del modelo de constituyentes obtendrá las representaciones en el formato de dependencias.

1.3 Organización de la memoria

La memoria consta de los siguientes capítulos:

- Estado del arte se introducen una serie de conceptos clave y el marco actual en el que esta agregado este proyecto.
- Diseño se proporciona un diseño de las herramientas implementadas durante el proyecto.
- Desarrollo se explican las estructuras de las herramientas y los algoritmos más relevantes que las mismas utilizan.
- Pruebas muestra las pruebas que certifican las herramientas creadas.
- Calidad de software se muestran los análisis de calidad obtenidos mediante diferentes herramientas.

¹ Stanford Dependencies: <https://nlp.stanford.edu/software/stanford-dependencies.shtml>

² Universal Dependencies: <http://universaldependencies.org>

2 Estado del arte

2.1 Análisis sintáctico

Como anteriormente se ha comentado, el análisis sintáctico consiste en analizar la estructura que presentan las palabras en las oraciones en una lengua, para obtener reglas, principios y normas que dan lugar a la combinación de las mismas.

El análisis sintáctico en sí mismo no tiene sentido si no hay una correspondencia con el significado que transmiten: la semántica oracional nos permite conocer los eventos y participantes de los enunciados en los actos comunicativos. En ocasiones y tareas parciales (como el reconocimiento de habla o el reconocimiento de términos y entidades) podemos prescindir del análisis sintáctico. Sin embargo, en otras tareas que sí lo requieran es necesario llegar a conocer relaciones estructurales entre elementos como: qué, quién, etc. Por tanto, la conclusión es que la sintaxis requiere de conocimientos de la morfología y de la semántica.

Actualmente existen dos representaciones sintácticas, que posteriormente han dado lugar a los analizadores sintácticos (software que intenta replicar las funciones realizadas por los lingüistas en el análisis sintáctico), modelo de constituyentes y modelo de dependencias.

En los analizadores sintácticos actuales hace uso de dos procesos denominados *Text Segmentation* (*segmentación del texto*) y *Morphological Analysis* (*análisis morfológico*), que han debido particularizarse para diferentes idiomas debido a que presentan algunas diferencias respecto al inglés, idioma utilizado como primera opción para el desarrollo de estos analizadores.

La segmentación del texto es la parte desarrollada en el software encargada de encontrar los límites entre palabras. En idiomas como el inglés, el fin de una palabra y otra viene dado por espacios o signos de puntuación. Sin embargo, en el chino no es posible hacer esa identificación pues no usan espacios para separar las palabras. Aunque este ejemplo se puede generalizar, ya que la segmentación es más sencilla en el caso de las lenguas occidentales comparado con las lenguas asiáticas.

El otro proceso comentado es el análisis morfológico, algo que no es tenido muy en cuenta en inglés ya que es un idioma en el que esta característica apenas aparece. Este análisis se lleva cabo en lenguas como el ruso, que es un idioma muy inflexible, pues está basado en declinaciones, es decir, la morfología puede ser un indicador de género, número, plural, etc.

Una declinación es una variación morfológica de las palabras para expresar distintas relaciones gramaticales dentro de una oración, su objetivo es marcar algunas relaciones sintácticas dentro de las oraciones como el sujeto, el objeto directo o el indirecto.

Por ejemplo, en ruso, la declinación consta de seis casos gramaticales que dependen si se refiere a un elemento masculino, femenino o neutro, singular o plural.

Un ejemplo de la primera declinación de ruso, en el que se puede apreciar como una misma palabra tiene una terminación diferente dependiendo del caso al que se refiera, ya sea genitivo (género), nominativo, dativo, etc.

Cases	Singular nouns endings	Examples *				
Nominative	-а, -я, -ия	трав <u>а</u>	земл <u>я</u>	юнош <u>а</u>	хим <u>ия</u>	Валер <u>ия</u>
Genitive	-ы, -и	трав <u>ы</u>	земл <u>и</u>	юнош <u>и</u>	хим <u>ии</u>	Валер <u>ии</u>
Dative	-е, -и	трав <u>е</u>	земл <u>е</u>	юнош <u>е</u>	хим <u>ии</u>	Валер <u>ии</u>
Accusative	-у, -ю	трав <u>у</u>	земл <u>ю</u>	юнош <u>у</u>	хим <u>ию</u>	Валер <u>ию</u>
Instrumental	-ой, -ей	трав <u>ой</u>	земл <u>ей</u>	юнош <u>ей</u>	хим <u>ией</u>	Валер <u>ией</u>
Prepositional	-е, -и	(о) трав <u>е</u>	(о) земл <u>е</u>	(о) юнош <u>е</u>	(о) хим <u>ии</u>	(о) Валер <u>ии</u>

Figura 1. Ejemplo de la primera declinación del ruso³

2.1.1 Constituyentes

En este modelo la oración se fragmenta en constituyentes inmediatos, de forma que en el análisis sintáctico se pueden reconocer diferentes unidades sintácticas, como palabras, sintagmas, cláusulas u oraciones.

Un constituyente sintáctico es una palabra, o una agrupación de palabras, que funciona en conjunto como una unidad dentro de la estructura jerárquica de una oración. Un constituyente puede descomponerse frecuentemente en dos subsecuencias o más, cada una de las cuales es, a su vez, otro constituyente. El conjunto de todos los constituyentes de una oración es un conjunto parcialmente ordenado, en donde el orden se basa en la descomponibilidad de los constituyentes en otros.

Algo a destacar del modelo de constituyentes es que los sintagmas de esta gramática deben tener obligatoriamente un núcleo y que ese sintagma tenga la misma denominación que su núcleo, es decir, en un sintagma verbal no puede haber un núcleo que sea un adjetivo, debe ser un verbo.

La gramática de constituyentes tiene una representación en forma de árbol, en el cual se muestra las funciones gramaticales como categorías por su posición dentro el éste. La representación en árbol es una de las más usadas, pues muestra de forma jerárquica las relaciones entre los constituyentes. Más formalmente, un árbol sintáctico es un grafo que representa esta relación de orden parcial. Otro tipo de representación es mediante una estructura encadenada de corchetes o paréntesis, en la que es más complicado distinguir la composición de los constituyentes.

La representación en forma de árbol parte de un constituyente principal, que sería la propia oración, para después realizar una división de la misma en sintagma nominal, sujeto y sintagma verbal, predicado. Esta estructura es muy común en algunos idiomas, como el inglés, motivo por el cual el modelo de constituyentes es tan bueno para representar sintácticamente esta lengua, mientras que para lenguas con una morfología más rica han tenido peores resultados, como pueden ser el ruso o el chino. Está claro que el gran número de formas flexionadas, la libertad de colocación de las palabras y el uso de información morfológica para indicar relaciones gramaticales y funciones sintácticas hacen que el análisis sintáctico de estas lenguas sea mucho más difícil, en comparación con el inglés, ya que el este modelo funciona muy bien en este idioma debido a su estructura constante y regular.

³ Declinaciones del ruso: <http://masterrussian.com/aa052000a.shtml>

En el inglés siempre encontramos la estructura compuesta por sujeto (NP) y predicado (VP). El siguiente ejemplo es una oración de ejemplo sencilla que utiliza *Stanford Parser* en su versión online. En ella se puede apreciar la estructura comentada.

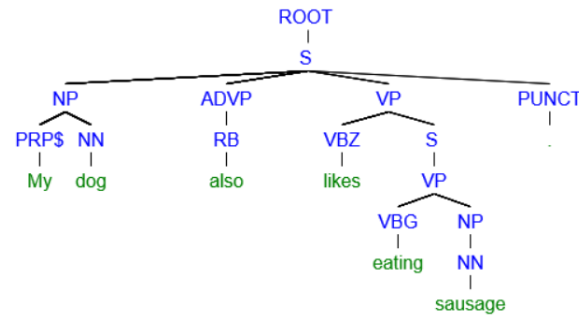


Figura 2. Ejemplo de árbol de constituyentes en inglés mediante la herramienta de Stanford.

A continuación, tenemos varios ejemplos obtenidos del treebank de constituyentes en español creado por el departamento de lingüística de la UAM y que se utilizarán posteriormente para la transformación a dependencias.

En el primer árbol, se puede apreciar que la estructura es igual en el sentido que existe un constituyente *NPSUBJ*, que indica que es el sujeto, y otro *VPTENSED*, que compone el predicado, es decir, sigue la misma estructura que la oración en inglés de la **Figura 2**.

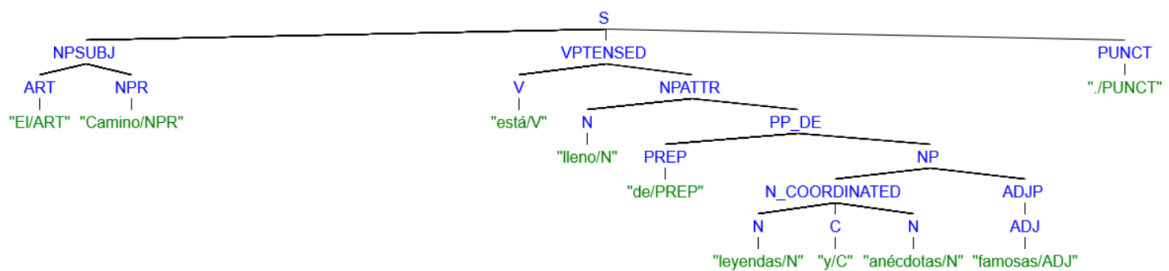


Figura 3. Primer ejemplo de árbol de constituyentes en español.

En este ejemplo se puede apreciar una perfecta diferenciación entre los dos constituyentes principales, además de seguir la estructura más sencilla, comentada anteriormente. En la parte del predicado se puede apreciar como un constituyente no solo puede estar formado por elementos finales, es decir, aquellos que no dan lugar a nuevas subsecuencias. Se puede apreciar como los diferentes constituyentes se van encadenando unos dentro de otros respetando sus dependencias hasta llegar finalmente a los elementos terminales que contienen las palabras que componen la oración.

A continuación, otro caso que se puede encontrar en el treebank. En este árbol se puede observar que ambos términos *NPSUBJ* y *VPTENSED* están completos, sin embargo, los elementos se encuentran invertidos. En inglés no se da este caso en el que primero se encuentre el predicado y después el sujeto, para este caso el analizador sintáctico puede llegar a tomar el sujeto como algún complemento del predicado, algo que no es correcto.

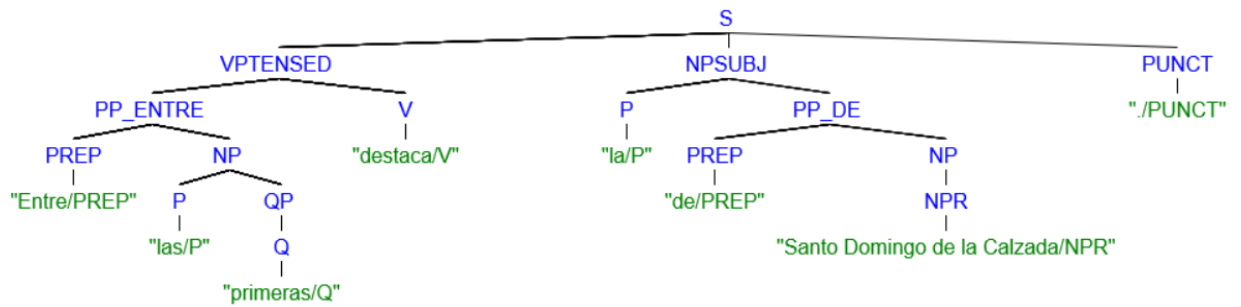


Figura 4. Segundo ejemplo de árbol de constituyentes en español.

2.1.2 Dependencias

Las relaciones de dependencias se establecen conectando las palabras individuales con su núcleo inmediatamente superior. La dependencia se establece como una relación binaria asimétrica, pues parte del núcleo hasta sus dependientes. La representación más usada es la de un grafo dirigido etiquetado con la función del elemento dependiente.

Si recordamos el modelo de constituyentes, se basaba en reconocer las diferentes unidades sintácticas, es decir, palabras, sintagmas, cláusulas u oraciones, mientras que en dependencias las relaciones se establecen únicamente entre palabras.

Las gramáticas de dependencias por el contrario, a las de constituyentes, no asumen esta estructura de árbol, optando por que el verbo sea la raíz de la oración donde todos los demás términos están directa o indirectamente relacionados con el verbo. Por lo que las gramáticas de dependencias no contienen el sintagma verbal como tal, sino las relaciones directas con el verbo, lo que las hace más adecuadas para lenguas con orden libre de constituyentes, como puede ser el ruso o chino.



Figura 5. Árbol de dependencias en español

La gramática de dependencias muestra las relaciones entre las palabras como funciones sintácticas, sujeto, objeto directo, etc. Por el contrario, en constituyentes se representan las funciones gramaticales como categoría derivadas por su posición en el árbol.

Actualmente se asume que las gramáticas de dependencias están mejor equipadas para tratar estructuras de las lenguas con orden libre de constituyentes y rica morfología, porque su formato de representación no se basa en la posición de las palabras y en su agrupamiento interno en sintagmas, como lo hacen las gramáticas de constituyentes

Desde la perspectiva computacional, la representación de dependencias es más eficiente que la de constituyentes, porque su estructura es más básica y restringida: básicamente solo hay que construir la relación entre el núcleo y sus dependientes, sin la complejidad de construir profundas ramificaciones de constituyentes.

2.2 Stanford Dependencies

El Stanford Dependencies es un analizador sintáctico estadístico que proporciona una representación gramatical de las relaciones entre dos palabras, dentro de una oración, para ello evalúa la frase entera y obtiene todas las relaciones entre elementos siempre dos a dos.

La tipología desarrollada por la Universidad de Stanford pretende proporcionar una estructura simple para la descripción de las relaciones gramaticales dentro de una oración y que pueda entenderse incluso por personas que no tengan experiencia en el área de la lingüística.

En *Stanford Dependencies* lo que se intenta es proponer una taxonomía mejorada, pudiendo reconsiderar algunas de las decisiones que se tomaron originalmente en el desarrollo de las dependencias. Se sugiere una taxonomía que tiene en su núcleo un conjunto de relaciones gramaticales, completadas en algunas ocasiones con subtipos para relaciones particulares del lenguaje, pues no se puede generalizar todo para todos los idiomas, es decir, que en algunos casos es necesario concretar la funcionalidad asociada a los elementos añadiendo información que complemente la categorización realizada. En *Stanford Dependencies* se pueden observar algunos casos en los que la etiqueta tiene la siguiente estructura:

nombre_relación:subtipo_relación (principal - posición, dependiente - posición)

Una parte poco elaborada en los primeros diseños (2008) es que se adopta la hipótesis lexicalista en la sintaxis, por medio de la cual las relaciones gramaticales deben estar entre palabras o lexemas. Hay un largo debate no resuelto entre teorías que intentan construir palabras y frases usando los mismos mecanismos sintácticos frente a teorías en las que la palabra es la unidad fundamental, que ven los procesos morfológicos que construyen palabras como fundamentalmente ocultas y diferentes a las que construyen oraciones.

Finalmente se usa la relación entre lexemas para determinar la relación entre ellas y así poder crear un adecuado conjunto de etiquetas de funcionalidad para los lenguajes en los que se está desarrollando este proyecto.

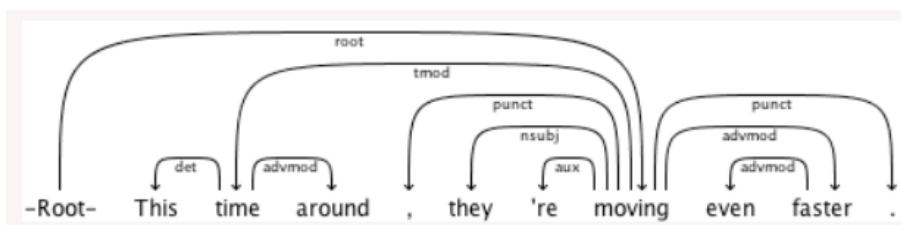


Figura 6. Árbol de dependencias de Stanford⁴

Además, se han usado algunas herramientas desarrolladas por la propia Universidad de Stanford para otros proyectos y que han tenido utilidad, como son el *Shift-Reduce Constituency Parser*, así como otras que son exclusivas de esta parte de dependencias, *Neural Network Dependency Parser*.

Shift-Reduce Constituency Parser mantiene un estado del árbol que se ha analizado con las palabras de la oración en una cola y los árboles parcialmente terminados en una pila, aplicando las diferentes transiciones al estado actual del árbol hasta que la cola esté vacía y la pila actual sólo contenga un árbol terminado.

El estado inicial es tener todas las palabras en orden en la cola, con una pila vacía. Las transiciones que se pueden aplicar son:

⁴ Red neuronal de Dependency Parser Stanford: <https://nlp.stanford.edu/software/nndep.shtml>

- SHIFT
- UNARY REDUCE
- BINARY REDUCE
- FINALIZE
- IDLE

Las transiciones se determinan actualizando el estado actual y usando un perceptrón multiclasa para determinar la próxima transición.

El entrenamiento se realiza iterando sobre los árboles repetidamente hasta que se alcanza cierta convergencia. Lo normal es comenzar desde el estado básico y aplicar las transiciones elegidas por el analizador hasta que se obtiene una incorrecta y ya no se puede reconstruir el árbol. Las características usadas en el momento de la decisión incorrecta tienen sus pesos ajustados, con la transición correcta consiguiendo los pesos de característica aumentados y la transición incorrecta disminuyó.

Neural Network Dependency Parser, este parte del analizador de *Stanford Dependencies* solo se encuentra en desarrollo para dos lenguas, el inglés y el chino. El funcionamiento se basa en una exploración de tiempo lineal sobre las palabras de una oración. En cada paso se mantiene un *parse* parcial, una pila de palabras que se están procesando y un buffer con aquellas palabras que faltan por procesar. El analizador aplica continuamente transiciones a su estado antes de que el buffer se vacíe y el grafo de dependencias quede completo.

El estado inicial es tener todas las palabras ordenadas en el buffer, con un nodo ROOT en la pila. Se pueden aplicar las siguientes transiciones:

- LEFT-ARC
- RIGHT-ARC
- SHIFT

El analizador decide entre transiciones en cada estado usado un clasificador de red neuronal. Representaciones del estado actual del analizador son proporcionadas como entradas a este clasificador, el cual elige entre las posibles transiciones.

El clasificador que alimenta el analizador se entrena y para ello toma cada una de las oraciones usadas para el entrenamiento y genera muchos ejemplos que indican la transición que debe hacerse en cada estado para alcanzar el análisis final correcto. Finalmente, la red neuronal se entrena con estos ejemplos.

2.3 Google Dependencies

Con el propósito de aprender el lenguaje natural, Google también ha entrado en este tema creando un software de procesamiento del lenguaje extendido para multitud de lenguas, alrededor de unas setenta, es lo que han denominado *Google SyntaxNet*.

SyntaxNet es un software que toma una frase como entrada, para la cual realiza un etiquetado de cada una de las palabras de la oración con una categoría gramatical (*part-of-speech tag*), esta categoría describe la función sintáctica de la palabra y permite establecer las relaciones sintácticas entre dos palabras de una misma frase con la representación de árbol de dependencias.

Pero uno de los problemas más comunes en el lenguaje natural es la ambigüedad, por ello, los analizadores del lenguaje natural deben tener en cuenta las alternativas y encontrar la estructura que mejor se adecue el contexto de la oración.

Con el ejemplo, *Alice drove down the street in her car* (*Alice condujo por la calle en su coche*), Google con su modelo de dependencias es capaz de encontrar dos posibles análisis.

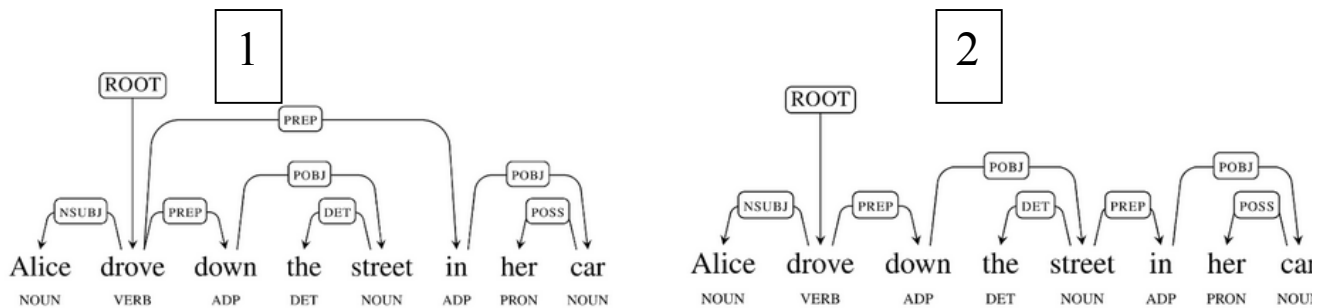


Figura 7. Árboles de dependencia de Google⁵

En el árbol izquierdo, se obtiene la interpretación de que Alice está conduciendo su coche, mientras que la segunda se podría tener un significado absurdo, como que la calle se encuentra situada en su coche. La ambigüedad viene dada por la preposición *in*, que puede modificar a *car* o a *street*. La ambigüedad suele darse en casos más complejos que este, pues hay mucha combinación de elementos.

La resolución de *SyntaxNet* es la aplicación de redes neuronales. La oración de entrada se va procesando elemento a elemento, aplicándose reglas sucesivamente, teniendo en cuenta los elementos ya analizados, pues la dependencia entre esos elementos puede establecer una funcionalidad u otra, de ahí la ambigüedad.

También es importante decir que *SyntaxNet* hace uso de los procesos de segmentación de texto y análisis morfológicos, los cuales se han tenido que adaptar para diferentes lenguas, ya que no se puede reutilizar directamente los desarrollos realizados para el inglés.

Para finalizar comentar que el método de entrenamiento consiste en entrenar muchos modelos en paralelo a través del algoritmo, y cuando uno obtiene buenos resultados, se entrena a los demás modelos con opciones similares para afinar los resultados.

2.4 Universal Dependencies

El formato de *Universal Dependencies* se basa en una visión sintáctica del lenguaje, lo que significa que las relaciones de dependencia son entre términos, es decir, palabras.

Universal Dependencies es un proyecto en desarrollo en el que se pretende crear un treebank sólido para varios idiomas, con el objetivo de facilitar el aprendizaje y análisis desde la tipología lingüística.

El esquema que representa las dependencias está basado en la evolución a escala global de *Stanford Dependencies*, *Google Universal part-of-speech tags* e *Intersect Interlingua* para etiquetas morfológicas.

El propósito fundamental de este proyecto es crear un inventario universal de etiquetas para las relaciones entre elementos y reglas, que permitan establecer para numerosos idiomas esas mismas etiquetas. Es necesario tener en cuenta las relaciones entre palabras en los diferentes idiomas, pues

⁵ Google Research Blog: <https://research.googleblog.com/2016/05/announcing-syntaxnet-worlds-most.html>

no todos presentan una estructura similar, lo que implica que habrá reglas más genéricas que puedan servir en muchos de ellos, y otras que será necesario particularizar para cada idioma.

2.5 Formato de representación

En el estudio de todos los modelos se han encontrado dos formatos de representación de las relaciones de dependencia a parte de los grafos dirigidos comentados en apartados previos.

2.5.1 Formato Stanford

Es un formato desarrollado por la Universidad de Stanford que representa las relaciones establecidas entre palabras. Existen cinco variantes, pero todas en definitiva siguen el mismo formato.

Cada relación se representa en texto plano como:

nombre_relación (principal - posición, dependiente - posición)

Tanto el principal como el dependiente son palabras de la oración junto con el índice que indica la posición que ocupa ese término dentro de la frase. Esta representación también se ha adaptado al formato XML, el cual nos recuerda más a un árbol.

La diferencia entre las cinco variantes existentes es como se nombra a la funcionalidad o etiqueta de relación entre el término principal y el dependiente, pues para Stanford existen diversas formas de representar esa etiqueta, dependiendo de cuanta información se quiera mostrar en ésta.

Ahora se mostrará un ejemplo con el formato básico para la siguiente frase, junto con la representación en forma de grafo:

*"Bell, a company which is based in LA, makes and distributes computer products."*⁶

```
nsubj(makes-11, Bell-1)
det(company-4, a-3)
appos(Bell-1, company-4)
nsubjpass(based-7, which-5)
auxpass(based-7, is-6)
rcmod(company-4, based-7)
prep(based-7, in-8)
pobj(in-8, LA-9)
root(ROOT-0, makes-11)
cc(makes-11, and-12)
conj(makes-11, distributes-13)
nn(products-15, computer-14)
dobj(makes-11, products-15)
```

⁶ Manual Stanford Dependencies: https://nlp.stanford.edu/software/dependencies_manual.pdf

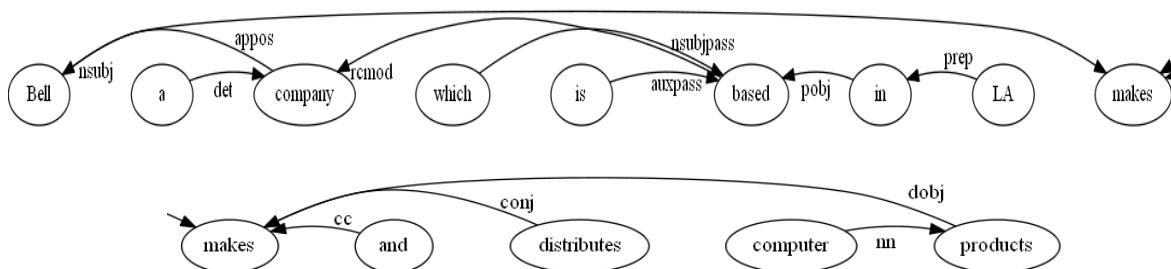


Figura 8. Representación del ejemplo de *Stanford Dependencies* generada con Graphviz

Cada una de las relaciones indica la funcionalidad como primer elemento que se muestra, después entre paréntesis se encuentran los elementos relacionados, primero se muestra la palabra, después la posición que ocupa ésta dentro de la oración, es necesario tener en cuenta los signos de puntuación, en el ejemplo, *makes* tiene como índice el 11 porque es su lugar en la frase, habiendo contado previamente las dos comas.

2.5.2 Formato CoNLL

Esta representación de las relaciones de dependencias ha sido creada por uno de los proyectos citados anteriormente, *Universal Dependencies*. Además, es necesario decir que el trabajo de Google utiliza este formato y que Stanford llega a utilizarlo también, aunque este da preferencia al tipo de representación que ellos desarrollaron.

Consiste en generar por cada oración una pequeña tabla sobre las relaciones que existen en ella. A diferencia del formato usado por la Universidad de Stanford, se incluyen otros campos, que dan información adicional sobre cada una de las palabras.

Los campos de los que consta esta tabla son:

- ID: posición que ocupa la palabra dentro de la oración
- FORM: forma de la palabra
- LEMMA: lema de la forma de la palabra
- UPOSTAG: etiqueta universal part-of-speech
- XPOSTAG: etiqueta propia de cada idioma
- FEATS: lista de características morfológicas proporcionadas por el inventario universal o que defina la propia lengua.
- HEAD: posición del elemento con el que se establece la relación
- DEPREL: etiqueta que indica la relación de dependencia establecida con el elemento indicador por *HEAD*.

A continuación, un ejemplo con el formato establecido por *Universal Dependencies*, para la oración: "They buy and sell books."

#	text = They buy and sell books.						
1	They	they	PRON	PRP	Case=Nom Number=Plur	2	nsubj
2	buy	buy	VERB	VBP	Number=Plur Person=3 Tense=Pres	0	root
3	and	and	CONJ	CC	—	4	cc
4	sell	sell	VERB	VBP	Number=Plur Person=3 Tense=Pres	2	conj
5	books	book	NOUN	NNS	Number=Plur	2	obj
6	.	.	PUNCT	.	—	2	punct

Figura 9. Representación de las relaciones de dependencias con formato CoNLL de *Universal Dependencies*⁷

Aquí se puede ver la representación en forma de grafo del resultado tras aplicar el modelo de gramática de dependencias.

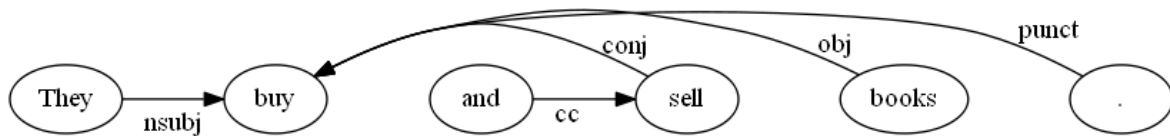


Figura 10. Representación de grafo del ejemplo de Universal Dependencies generada con Graphviz

⁷ Formato Universal Dependencies: <http://universaldependencies.org/format.html>

3 Diseño

3.1 Análisis

Para la implementación ha sido necesario estudiar la estructura que presentan los árboles de constituyentes que forman el treebank, pues para poder hacer la transformación de un modelo a otro se tendrán que recorrer estos árboles, para poder acceder a cada uno de los elementos que lo componen.

```
[S
  [NPSUBJ
    [ART "La/ART"]
    [N "policía/N"]]
  [VP_TENSED
    [V "descubre/V"]
    [NPOBJ1
      [ART "un/ART"]
      [ADJP
        [ADJ "gran/ADJ"]]
        [N "arsenal/N"]
      ]
      [PP_DE
        [PREP "de/PREP"]
        [NP
          [NPR "ETA/NPR"]]]]
    [PP_EN_LOCATIVE
      [PREP "en/PREP"]
      [NP
        [NPR "Francia/NPR"]]]
    [PP_TRAS_TIME
      [PREP "tras/PREP"]
      [CL_INFINITIVE
        [VPUN_TENSED_INFINITIVE
          [V "producirse/V"]
          [NPSUBJ
            [ART "un/ART"]
            [N "incendio/N"]]
          [PP_EN_LOCATIVE
            [PREP "en/PREP"]
            [NP
              [ART "un/ART"]
              [N "chalé/N"]]]]]]]]]
  [PUNCT "."/PUNCT"]]
```

Figura 11. Árbol de constituyentes de ejemplo

Aunque no solo se ha estudiado la composición del árbol sino también los distintos tipos de estructuras que pueden componer una frase y como pueden estar unos incluidos dentro de otros.

A continuación, se muestra un árbol de constituyentes bastante completo sobre el que se explicarán cómo son algunas de las estructuras.

Para facilitar la visualización del árbol, se incluye una imagen con la estructura propia del árbol de la oración de ejemplo.

"La policía descubre un gran arsenal de ETA en Francia tras producirse un incendio en chalé."

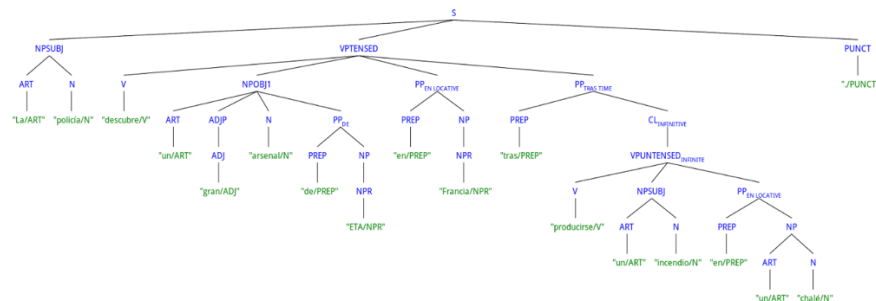


Figura 12. Representación del árbol de constituyentes de ejemplo

Supongamos que la oración tenga la estructura más básica compuesta por sujeto y predicado, que es lo que hace que la gramática de constituyentes sea tan buena para el inglés. Esa estructura que

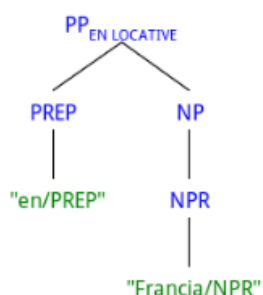
está presente en la oración puesta de ejemplo permite que puedan apreciarse dos grandes subárboles, uno *NPSUBJ*, que sería el equivalente al sujeto, mientras que el *VPTENSED* está referido al predicado, por último, se puede apreciar a la derecha del todo un único elemento que representa el fin de la oración.

Debido a que esta frase no es de las más complejas que hay en el treebank, se ve un sujeto muy sencillo, compuesto únicamente por un determinante y un sustantivo, pero no todos los ejemplos son como este, hay algunos en los que el sujeto se encuentra compuesto por varias oraciones, puede tener un complemento preposicional o incluso se dan casos en los que no existe el sujeto.

De la misma manera, la estructura que presenta el predicado donde se puede apreciar más elementos. Dentro del predicado se ve que el primer nodo está compuesto por el verbo, que en este caso presenta una forma simple, por lo que este nodo solo tiene un hijo que cuelgue de él, cuando se dan casos de tiempos verbales compuestos se tiene que el nodo *V* se expande para representar esa forma verbal.

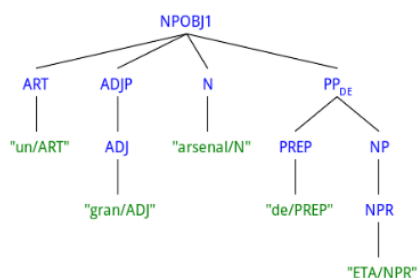
A continuación del verbo se tiene un *NPOBJI*, que en este caso representa lo que se denomina como complemento directo en sintaxis. Este tipo de complementos siempre debe tener un elemento nominal, es decir, un sustantivo *N*, que dependa directamente del anterior, pues sino el elemento superior no podría contener la partícula *NP*.

Con este pequeño ejemplo se puede deducir una regla genérica para cualquier estructura compuesta de la oración. Siempre que aparezca una categoría en el nodo raíz de un árbol o subárbol, este debe contener un elemento que dependa directamente de la raíz y que tenga la misma categoría que él. En la imagen se pueden apreciar algunos ejemplos.



Este ejemplo se trata de un sintagma preposicional de lugar, la representación del tipo de sintagma es PP seguido del tipo que le corresponda (tiempo, lugar, ...). Al ser un complemento preposicional, como el propio nombre indica, debe haber una preposición, el cual es hijo directo del sintagma.

Figura 13. Sintagma preposicional



En este caso tenemos un árbol más completo, pero se va a cumplir lo mismo que en el caso anterior, pero esta vez para un sustantivo, pues el nodo padre contiene NP, que indica sintagma nominal.

Figura 14. Sintagma nominal

Tanto la teoría como la experiencia de haber revisado muchas oraciones nos dice que no puede existir un tipo de sintagma concreto que no contenga al menos un elemento de ese mismo tipo. Además de las muestras anteriores, también existe igualmente para sintagmas de tipo verbal, adverbial, adjetival, etc.

3.1.1 Oraciones y constituyentes

Algunas de estructuras no son tan sencillas como las que aparecen en los ejemplos previos, pues estos son constituyentes muy comunes, pero además existen otros componentes con estructuras

más complejas y que en algunas ocasiones condicionan el tipo de la oración, es decir, su aparece la palabra *COORDINATED* implica que la oración es coordinada.

Las cláusulas no terminales, es decir, aquellas que forman un subárbol dentro de la oración siguen una estructura determinada, primero se indica el tipo de sintagma del que se trata, por ejemplo, *N* o *NP* para sintagmas nominales, seguido de guión bajo y una especificación del sintagma, por ejemplo, *COORDINATED*, dando lugar a *N_COORDINATED*, que indica que es una cláusula de sintagma nominal en la que los nombres de tal sintagma se encuentran unidos por una relación de coordinación, es decir, la preposición *y*.

Este ha sido un pequeño ejemplo aplicado a los sintagmas nominales pero igualmente se aplica a sintagmas verbales, adjetivales o adverbiales, oraciones subordinadas, etc.

CLÁUSULA	DESCRIPCIÓN	ESPECIFICACIÓN
(DATE)	Fecha	
(HOUR)	Hora	
(N_COORDINATED)	Sintagma nominal	Sustantivos coordinados
(NP_COMPARATIVE-1)	Sintagma nominal	Comparativo
(PP_A_COORDINATED)	Complemento preposicional	preposiciones coordinadas
(PP_A_LOCATIVE)	Complemento preposicional	Preposición de lugar
(NPOBJ1)	Objeto directo	
(NPOBJ2)	Objeto indirecto	
(NPSUBJ_COMPARATIVE)	Sintagma nominal sujeto	Comparativo
(NPSUBJ_COORDINATED)	Sintagma nominal sujeto	Sujetos coordinados
(QP)	Sintagma cuantitativo	
(V_COORDINATED)	Verbos coordinados	
(VPTENSED_COORDINATED)	Sintagma verbal	Verbos coordinados
(VPTENSED_DISTRIBUTIVE-1)	Sintagma verbal	Verbos distributivos
(VPTENSED_PASSIVE_COORDINATED)	Sintagma verbal	Verbos pasivos coordinados
(VPTENSED_PASSIVE)	Sintagma verbal	Verbo pasivo
(VPUNTENSED_GERUND)	Sintagma verbal impersonal	Gerundio
(VPUNTENSED_INFINITE_COORDINATED)	Sintagma verbal impersonal	Infinitivo coordinado
(VPUNTENSED_INFINITE_PASSIVE)	Sintagma verbal impersonal	Infinitivo pasivo
(VPUNTENSED_INFINITE)	Sintagma verbal impersonal	Infinitivo
(ADVP_COMPARATIVE)	Sintagma adverbial	Adverbio comparativo
(ADVP_INTERROGATIVE)	Sintagma adverbial	Adverbio interrogativo
(ADVP_LOCATIVE)	Sintagma adverbial	Adverbio de lugar
(ADVP_NEG)	Sintagma adverbial	Adverbio de negación
(ADVP_TIME)	Sintagma adverbial	Adverbio de tiempo
(CL_COMPARATIVE)	Oración subordinada	Oraciones comparativas
(CL_COMPLETIVE)	Oración subordinada	Oraciones completivas
(CL_CONDITION)	Oración subordinada	Oraciones condicionales
(CL_COORDINATED)	Oración subordinada	Oraciones coordinadas

Tabla 1. Tabla resumen de cláusulas de constituyentes

En relación con la **Tabla 1** y antes de continuar sería bueno recordar los diferentes tipos de oraciones que existen y que partículas las identifican dentro del treebank:

- Copulativas: oraciones que contienen como verbo principal alguno de las formas verbales de ser, estar y parecer. Estas oraciones no tienen ninguna etiqueta específica que las diferencie, por lo que ha sido necesario hacer una búsqueda manual en un fichero que contiene el treebank original creado por el departamento de lingüística que incluye información adicional como el lema, que en el caso de los verbos indica su tiempo infinitivo, por lo que han buscado las formas verbales de los verbos ser, estar y parecer. David Crystal dijo sobre el lema: *“Esencialmente es una representación abstracta que abarca todas las variaciones léxicas*

formales que pueden aplicarse”.⁸ Estas oraciones tienen una gran particularidad, el verbo copulativo no puede ser “root” de un árbol de dependencias, ya que este carece prácticamente de significado porque actúa como nexo entre el sujeto y el atributo, que puede ser un adjetivo o venir definido por una oración.

Debido a ello el “root” debe ser un adjetivo que lo condicione o en caso de no haberlo, se intentará encontrar el “root” en una oración subordinada, y si no fuese el caso se buscaría una raíz de carácter nominal, considerándose la oración como una frase sin verbo.

- Subordinada: son aquellas oraciones que dependen del núcleo de otra oración, por lo que actúa como un constituyente dentro de otro. Dentro del treebank se pueden identificar por la partícula *CL_XXX*, *XXX* hace referencia al tipo que puede ser esa oración subordinada.
- Coordinada: son oraciones compuestas por dos o más cláusulas independientes entre sí, unidas por alguna conjunción como es *y*. Se pueden encontrar marcadas por la etiqueta *XXX_COORDINATED*.

Como se puede ver en la **Figura 12**, existe una cláusula subordinada de tipo infinitivo, pues el verbo principal de esa oración se encuentra en ese tiempo verbal. La estructura que presenta la oración subordinada es estándar, con la particularidad de que es el verbo el que abre la oración y no el sujeto como suele ser lo más habitual o correcto.

Como ya se ha comentado anteriormente, el español es un idioma con gran flexibilidad léxica, lo que permite hacer múltiples combinaciones en el orden de los elementos, incluso hay casos en los que alguno de estos se puede llegar a obviar, como puede ser el sujeto.

Algunos ejemplos muy comunes y que utilizamos a menudo en nuestro día a día son la omisión del sujeto, lo que en lingüística se denomina sujeto elidido. También suele ser frecuente en nuestro idioma la alteración del orden de algunos elementos como el sujeto con el verbo o con el predicado, este ejemplo se puede apreciar muy bien en la **Figura 3**.

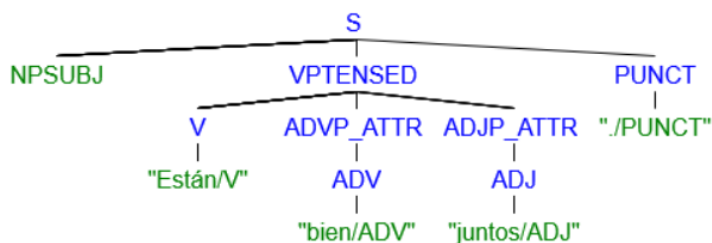


Figura 15. Árbol de constituyentes con sujeto elidido.

En el ejemplo se aprecia que el sujeto se encuentra vacío, es lo que antes se ha definido como sujeto elidido. En el treebank se puede reconocer normalmente mediante la etiqueta *NPSUBJ_ELI* o como se muestra en la figura, un *NPSUBJ* vacío.

3.1.2 Transformación de árboles de constituyentes

En el proceso de transformación de cada uno de los árboles de constituyentes se ha decidido dividirlo en dos partes, siempre y cuando la frase lo permita, lo que quiere decir que primero se aplicará el algoritmo para el sujeto con todos sus elementos y después en el predicado. Una vez que se ha realizado la transformación de ambas partes, se pasa a juntar ambos y terminar de establecer las relaciones pertinentes entre los elementos del sujeto que así lo requieran y el nodo raíz que compondrá el árbol de dependencias, es decir, el verbo principal de la oración.

⁸ Definición dada por David Crystal en el libro *Dictionary of Linguistics and Phonetics*, 2008

Para cada una de las partes comentadas anteriormente, que componen un árbol, se aplica una función recursiva de recorrido en profundidad hasta llegar a los nodos terminales.

Cuando se llega a un elemento final o terminal del árbol se crea la relación de ese elemento, sin llegar a completarse pues aún no se sabe el nodo con el que se establecerá ésta. Este paso se hace para cada uno de los elementos de un subárbol, y a la vez que se crea la relación se comprueba que ese elemento pueda ser un posible nodo raíz, en caso de ser así se marca ese nodo como tal.

Además, es necesario mantener un registro de por dónde se va pasando a medida que se recorre el árbol, este registro es lo que se llamará contexto y será la etiqueta asignada a las cláusulas que componen nuevo subárbol. Este contexto irá variando a medida que se avanza el recorrido del árbol, pero no se puede perder en ningún momento el registro de por dónde has pasado por lo que se guardará en una estructura de pila.

Cuando se han recorrido todos los elementos de un subárbol se completan las relaciones con aquella que se ha marcado como “root”. Después de establecer la relación con el “root” se puede pasar a completar la relación con la etiqueta de dependencia entre elementos, para ello hay que conocer el contexto en que se encuentra pues dos categorías asociadas a los elementos pueden tener una funcionalidad diferente dependiendo del contexto en el que se encuentren.

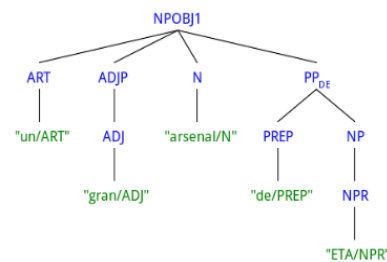


Figura 16. Subárbol que se usará para reflejar el seguimiento del algoritmo.

En la **Figura 16** se muestra un esquema de cómo se aplica el algoritmo al ejemplo que venimos manejando en este apartado.

```

NPOBJ1
  PILA-CONTEXTO: NPOBJ1
  un: no es root
  ADJP
    PILA-CONTEXTO: NPOBJ1, ADJP
    gran: es root del árbol cuyo padre es ADJP
    - completar árbol ADJP con el root: gran
    - desapilar contexto ADJP
    - PILA-CONTEXTO: NPOBJ1
  arsenal: es root del árbol cuyo padre es NPOBJ1
  PP_DE
    PILA-CONTEXTO: NPOBJ1, PP_DE
    de: no es root
    NP
      PILA-CONTEXTO: NPOBJ1, PP_DE, NP
      ETA: es root del árbol cuyo padre es NP
      - completar árbol NP con el root: ETA
      - desapilar contexto NP
      - PILA-CONTEXTO: NPOBJ1, PP_DE
    - completar árbol PP_DE con el root: ETA
    - desapilar contexto PP_DE
    - PILA-CONTEXTO: NPOBJ1
  - completar árbol NPOBJ1 con root: arsenal
  - desapilar contexto NPOBJ1
  - PILA-CONTEXTO:

```

En el esquema se ve cómo se va recorriendo el árbol. Se puede ver cómo se va pasando por todos los nodos, cuando llega a un nodo terminal como puede ser el caso de *un*, se hace la comprobación de “root”, la cual no se cumple en este caso. Lo mismo que se ha hecho en el caso de *un* se hace para los demás elementos finales, intentando encontrar algún elemento que pueda llegar a actuar como “root”, como es el caso de *gran*, *ETA* o *arsenal*.

Cuando se ha encontrado algún elemento “root” y se ha llegado al final del árbol, se completan las relaciones existentes dentro de este con el “root” correspondiente, exceptuando el propio “root”, pues este no puede estar relacionado consigo mismo, lo estará con otro elemento perteneciente a un árbol superior, además se completan las relaciones añadiendo la funcionalidad asociada a la par de categorías los elementos que componen la relación.

A medida que se van completando las relaciones y se va subiendo en el árbol, los “root” utilizados dejan de considerarse así, y algo similar ocurre con los contextos apilados, según se ha completado el árbol se descarta ese contexto, pues ya se pasaría al inmediatamente superior.

A continuación, un pequeño pseudocódigo sobre la función recursiva explicada:

```

FUNCION RECURSIVA (NODO N)
  IF (N ES TERMINAL)
    LISTA-RELACIONES.ADD(CREATE-RELACION(N))
  ELSE
    APILAR CONTEXTO
    FOREACH (CHILD: N.CHILDS())
      IF (CHILD ES TERMINAL)
        LISTA-RELACIONES.ADD(CREATE-RELACION(CHILD))
      ELSE
        LISTA-RELACIONES.ADD(FUNCION RECURSIVA (CHILD))

    COMPLETAR-RELACIONES(LISTA-REALCIONES)
    DESAPILAR CONTEXTO

  RETURN LISTA-RELACIONES

```

3.2 Limitaciones

A lo largo del desarrollo del algoritmo de transformación se han ido encontrando algunos problemas que debido a la estructura que presentan las oraciones del treebank en el modelo de constituyentes no permiten una correcta transformación a dependencias.

Como se comentó en el punto anterior hay diferentes tipos de oraciones, las que podemos considerar estándar, copulativas, subordinadas y coordinadas.

La transformación se empezó para las estructuras más estándar, es decir, oraciones sencillas con sujeto y predicado, y con complementos bastante sencillos.

A partir de esas oraciones se empezaron a detectar errores, lo que hizo que las oraciones que daban algún problema se revisasen a mano, yendo directamente al treebank de constituyentes, llegando a detectar algunos problemas que han creado limitaciones en la transformación a dependencias y se pretenden arreglar en un futuro, pero para ello es necesario que los lingüistas lo revisen y vean una mejor definición de algunos elementos en constituyentes que permitan el cambio.

Para empezar, las oraciones del grupo de las coordinadas no presentan transformación debido a que no ha sido posible por el momento, por parte de los lingüistas, establecer una correcta relación entre la conjunción que crea la unión de los elementos y éstos. Esto se debe a que los recursos utilizados como guía para la creación de las relaciones entre elementos no tienen unas reglas definitivas al respecto. Aún es un tema que se encuentra en pleno debate, pues hay varias opciones para hacerlo, pero es necesario que la referencia universal sea terminante para que se pueda aplicar correctamente.

Un dilema que surgió fue la aparición de los elementos elididos, es decir, los sujetos que no están presentes. La discusión en este caso se debe a que hay diferentes formas de tratarse, por lo cual es necesario llegar a una solución válida para todos los casos, algo a determinar por los lingüistas, por lo que se tomó la decisión con ellos de obviar ese elemento cada vez que aparezca de forma que no se crea ninguna relación con él, pasando así al siguiente elemento.

Otro problema encontrado tiene relación con las oraciones subordinadas, especialmente en aquellas que no presentan la cláusula de sujeto elidido, es decir, que tienen un sujeto. El error que se produce es debido al orden y la estructura que presentan estos constituyentes.

En las oraciones en las que es el verbo el que se encuentra en primer lugar y después aparecen el sujeto y los complementos, funciona correctamente, pues se va recorriendo recursivamente el árbol y detecta que el “*root*” de esa oración debe ser el verbo. Por el contrario, si es el sujeto el que está el primero, toma uno de sus elementos, normalmente un sustantivo como nodo padre, lo cual es incorrecto. En parte esto se debe a la estructura.

Luego se detectó otro caso que implica a la categoría de los pronombres. Esto se debe a que en un sintagma nominal (*NP*) puede no haber un elemento nominal como un sustantivo, pero sí un pronombre, el cual tendría la misma funcionalidad. El problema viene que no es posible tratar todos los pronombres existentes como núcleo o “*root*” de ese constituyente, ya que el pronombre *se* que en muchas ocasiones acompaña a los verbos para algunos tiempos verbales podría llegar a tomar el papel principal del verbo, haciendo una dependencia incorrecta, pues la buena debería ser con el verbo.

A continuación, un ejemplo en el que el “*root*” debería ser un pronombre:

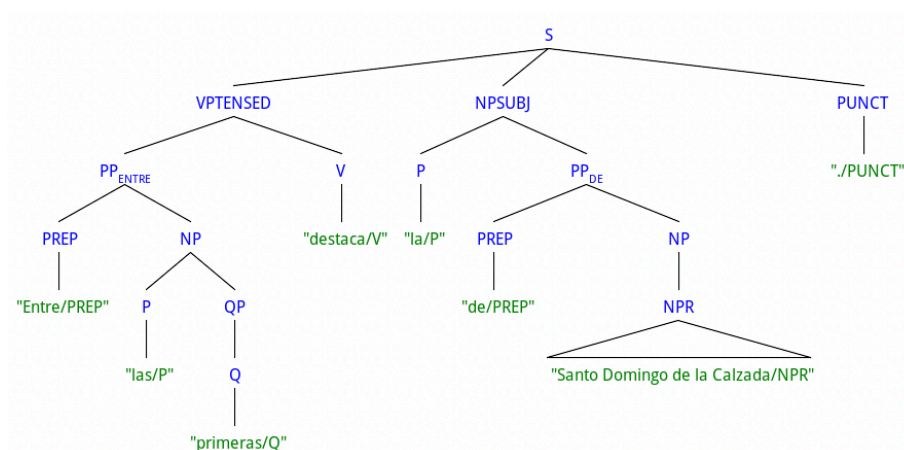


Figura 17. Árbol con problema de pronombre

En la imagen se puede apreciar mejor el problema, dentro del primer sintagma preposicional *PP_ENTRE*, hay un sintagma nominal que no contiene un sustantivo para poder hacer la función de “*root*” dentro de la cláusula *NP*, por lo que en este caso el pronombre *P* pasaría a ser el nodo padre. Como se ha comentado anteriormente, generalizar este caso de pronombres como raíz afectaría de forma negativa en los pronombres que tienen funcionalidad asociada al verbo.

4 Desarrollo

4.1 Plantilla en formato CoNLL

Un paso previo a la implementación del algoritmo fue la creación de la plantilla en formato CoNLL. El objetivo que tuvo la creación de esta plantilla facilitar la traducción que el departamento de lingüística de la Universidad Autónoma de Madrid debía hacer del treebank de constituyentes al nuevo treebank de dependencias.

La construcción de la plantilla ha consistido en crear un fichero Excel, en formato CSV, en el que cada una de las frases que componen el treebank ha sido tratada para dar lugar a una tabla como la que se puede apreciar en la **Figura 9**, pero con varias de las columnas vacías, cuya responsabilidad de rellenar pertenecía al departamento de lingüística.

Para llegar al formato deseado se partió de las frases originales del treebank, es decir, frases que no tienen formato de árboles de constituyentes pero que tienen una pequeña particularidad. Cada una de las palabras que componen la oración tiene asociada la categoría definida por el idioma, categoría que compondrá una de las columnas en el formato CoNLL.

A continuación, una frase del treebank original, en la que se puede apreciar como cada término tiene asociada su categoría, estando la palabra y la categoría separadas por un delimitador como es la barra inclinada:

```
La/ART policía/N descubre/V un/ART gran/ADJ arsenal/N de/PREP
ETA/NPR en/PREP Francia/NPR tras/PREP producirse/V un/ART
incendio/N en/PREP un/ART chalé/N ./PUNCT
```

La descomposición de la oración en las columnas se ha realizado teniendo en cuenta dos delimitadores, el primero el espacio que separa cada una de las palabras, por cada una de las ellas se aumenta un contador, el cual indica el índice o posición del término dentro de la oración. Una vez que se obtiene el grupo formado por palabra y categoría, se hace la división por el delimitador que es la barra inclinada.

Con las separaciones que se han mencionado, se obtienen tres de las columnas que componen la tabla, el índice, la palabra en sí y la categoría del idioma. De las demás columnas que componen la tabla, sólo alguna más se completa en esta parte del proyecto, la cual es la categoría universal. Cabe decir, que por el momento y hasta que se determine de forma definitiva por el departamento de lingüística, cual es la correspondencia más adecuada entre la categoría del idioma y la categoría universal, se ha decidido poner en la categoría universal (UPOSTAG) la misma que la del idioma (XPOSTAG).

ID	FORM	LEMMA	UPOSTAG	XPOSTAG	FEATS	HEAD	DREPEL
1	La	-	ART	ART	-	-	-
2	policía	-	N	N	-	-	-
3	descubre	-	V	V	-	-	-
4	un	-	ART	ART	-	-	-
5	gran	-	ADJ	ADJ	-	-	-
6	arsenal	-	N	N	-	-	-
7	de	-	PREP	PREP	-	-	-
8	ETA	-	NPR	NPR	-	-	-
9	en	-	PREP	PREP	-	-	-
10	Francia	-	NPR	NPR	-	-	-
11	tras	-	PREP	PREP	-	-	-
12	producirse	-	V	V	-	-	-
13	un	-	ART	ART	-	-	-
14	incendio	-	N	N	-	-	-
15	en	-	PREP	PREP	-	-	-
16	un	-	ART	ART	-	-	-
17	chalé	-	N	N	-	-	-
18	.	-	PUNCT	PUNCT	-	-	-

Figura 18. Plantilla en formato CoNLL en español

4.2 Formato de ficheros

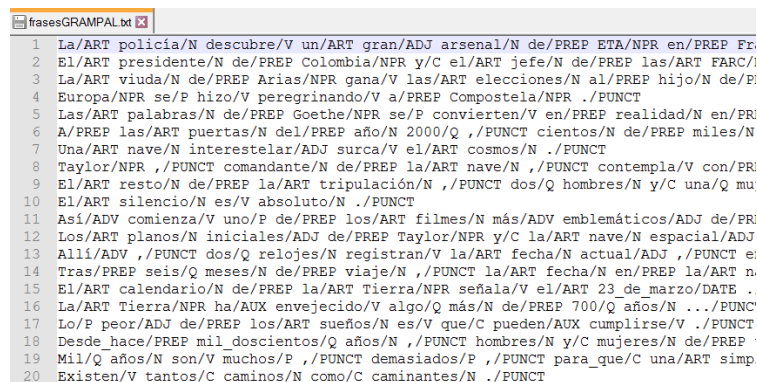
En este apartado se mostrará como son los ficheros utilizados a lo largo del proyecto, por un lado, los ficheros de entrada y por otro los de salida.

Como entrada se utilizan el treebank de constituyentes y un documento que contiene la tabla de etiquetas asociada los pares de elementos, mientras que para la salida se tienen dos posibles archivos, dependiendo del formato de salida escogido para representar el treebank de dependencias. Estos formatos han sido comentados en apartados anteriores, uno el formato Stanford y el otro sería el formato CoNLL.

4.2.1 Formato de entrada

El primer fichero que se usa es el que contiene las oraciones originales del treebank con su categoría asociada y que será utilizado en la generación del fichero de plantillas para las mismas frases. El formato utilizado en este fichero es texto plano, por lo que se escogió *txt*.

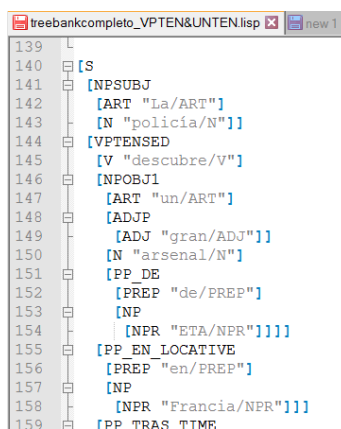
Este fichero se utiliza para generar la plantilla de oraciones en el formato de dependencias que los lingüistas debían rellenar con las relaciones de cada oración.



```
frasesGRAMPAL.txt
1 La/ART policía/N descubre/V un/ART gran/ADJ arsenal/N de/PREP ETA/NPR en/PREP Fr
2 El/ART presidente/N de/PREP Colombia/NPR y/C el/ART jefe/N de/PREP las/ART FARC/
3 La/ART viuda/N de/PREP Arias/NPR gana/V las/ART elecciones/N al/PREP hijo/N de/P
4 Europa/NPR se/P hizo/V peregrinando/V a/PREP Compostela/NPR ./PUNCT
5 Las/ART palabras/N de/PREP Goethe/NPR se/P convierten/V en/PREP realidad/N en/PRI
6 A/PREP las/ART puertas/N del/PREP año/N 2000/Q ./PUNCT cientos/N de/PREP miles/N
7 Una/ART nave/N interestelar/ADJ surca/V el/ART cosmos/N ./PUNCT
8 Taylor/NPR ./PUNCT comandante/N de/PREP la/ART nave/N ./PUNCT contempla/V con/PRI
9 El/ART resto/N de/PREP la/ART tripulación/N ./PUNCT dos/Q hombres/N y/C una/Q mu
10 El/ART silencio/N es/V absoluto/N ./PUNCT
11 Así/ADV comienza/V uno/P de/PREP los/ART filmes/N más/ADV emblemáticos/ADJ de/PRI
12 Los/ART planos/N iniciales/ADJ de/PREP Taylor/NPR y/C la/ART nave/N espacial/ADJ
13 Allí/ADV ./PUNCT dos/Q relojes/N registran/V la/ART fecha/N actual/ADJ ./PUNCT e
14 Tras/PREP seis/Q meses/N de/PREP viaje/N ./PUNCT la/ART fecha/N en/PREP la/ART n
15 El/ART calendario/N de/PREP la/ART Tierra/NPR señala/V el/ART 23_de_marzo/DATE .
16 La/ART Tierra/NPR ha/AUX envejecido/V algo/Q más/N de/PREP 700/Q años/N .../PUNC
17 Lo/P peor/ADJ de/PREP los/ART sueños/N es/V que/C pueden/AUX cumplirse/V ./PUNCT
18 Desde_hace/PREP mil_doscientos/Q años/N ./PUNCT hombres/N y/C mujeres/N de/PREP
19 Mil/Q años/N son/V muchos/P ./PUNCT demasiados/P ./PUNCT para_que/C una/ART simp
20 Existen/V tantos/C caminos/N como/C caminantes/N ./PUNCT
```

Figura 19. Fichero de oraciones originales en texto plano

El segundo archivo a utilizar es el que contiene todas las oraciones del treebank en forma de árboles de constituyentes, que serán la entrada del algoritmo y a partir de los cuales se hará la transformación a dependencias. Debido a que los árboles deben mantenerse balanceados se ha decidido usar el formato *lisp*.



```
treebankcompleto_VPTEN&UNTEN.lisp
139 L
140 [S
141 [NPSUBJ
142 [ART "La/ART"]
143 [N "policía/N"]
144 [VPTENSED
145 [V "descubre/V"]
146 [NPOBJ1
147 [ART "un/ART"]
148 [ADJP
149 [ADJ "gran/ADJ"]
150 [N "arsenal/N"]
151 [PP_DE
152 [PREP "de/PREP"]
153 [NP
154 [NPR "ETA/NPR"]]]]]
155 [PP_EN_LOCATIVE
156 [PREP "en/PREP"]
157 [NP
158 [NPR "Francia/NPR"]]]]]
159 [PP TRAS TIME
```

Figura 20. Fichero de oraciones como árboles de constituyentes

Por último, se hará uso del documento de relaciones que contiene todas las posibles de combinaciones de elementos dentro de un posible contexto junto con la etiqueta de funcionalidad

de dependencias asociada. La información que contiene se usará en el algoritmo de transformación para asignar correctamente las etiquetas de con la funcionalidad asociada las relaciones de dependencia que se vayan creando. Para estos datos se necesitaba un formato sencillo y estructurado, motivo por el que se ha escogido una hoja de cálculo, es decir, un documento Excel. Para la correcta lectura del fichero como hoja de cálculo de Excel ha sido necesario la incorporación de paquetes externos y específicos para este tipo de archivos.

1er elemento	2o elemento	contexto	etiqueta
ART	NPR	NPOBJ1	det
N	ADJ	NP	amod
NPR	ADJ	NP	amod
ADJ	N	NP	amod
ADJ	NPR	NP	amod
PREP	N	PP_DE	case

Figura 21. Fichero de relaciones

4.2.2 Formato de salida

Anteriormente en el apartado *Formato de representación* se ha hablado de dos formatos de representar las relaciones establecidas por el modelo de gramática de dependencias, uno es el desarrollado por la Universidad de Stanford y el otro sería el formato universal CoNLL.

La transformación de los árboles de constituyentes a árboles de dependencias puede dar lugar a dos representaciones diferentes, cada una correspondiente a los formatos comentados anteriormente.

Una vez aplicada la transformación a dependencias se hará el cambio de formato. Para ello se coge la estructura que compone ahora la oración, una lista de relaciones (*class Relation*). De la relación que compone ahora la frase se obtienen los elementos: *firstIndex*, *firstValue*, *secondIndex* y *relation*.

Algo que hay que destacar es que la variable *firstValue* está compuesta por la palabra y la categoría (*ejemplo: La/ART*), por ello será necesario hacer la separación por el delimitador y así poder sacar las columnas FORM, UPOSTAG y XPOSTAG.

ID	FORM	LEMMA	UPOSTAG	XPOSTAG	FEATS	HEAD	DEPREL
1	La		ART	ART			2 det
2	policía		N	N			3 nsubj
3	descubre		V	V			0 ROOT
4	un		ART	ART			6 det
5	gran		ADJ	ADJ			6 amod
6	arsenal		N	N			3 dobj
7	de		PREP	PREP			8 case
8	ETA		N	N			6 nmod
9	en		PREP	PREP			10 case
10	Francia		N	N			3 nmod
11	tras		PREP	PREP			12 case
12	producirse		V	V			3 advcl
13	un		ART	ART			14 det
14	incendio		N	N			12 dobj
15	en		PREP	PREP			17 case
16	un		ART	ART			17 det
17	chalé		N	N			12 nmod
18	.		PUNCT	PUNCT			3 punct

Figura 22. Formato CoNLL tras aplicar el algoritmo

El otro formato de salida disponible es el formato Stanford, que como se puede ver en el apartado *Formato Stanford*, las relaciones se muestran de forma más sencilla, equivalente a una tupla de elementos junto con la etiqueta de funcionalidad del primer elemento en relación al segundo.

4.3 Implementación

En este apartado describe los componentes que conforman el algoritmo implementando, y el comportamiento de los mismos en tiempo de ejecución. La **Figura 23** muestra la clase principal desde la cual se hace uso de las clases implementadas encargadas de la transformación de los

árboles, además de las clases que realizan la lectura y escritura de ficheros de salida del treebank en el formato de dependencias escogido (Stanford o CoNLL).

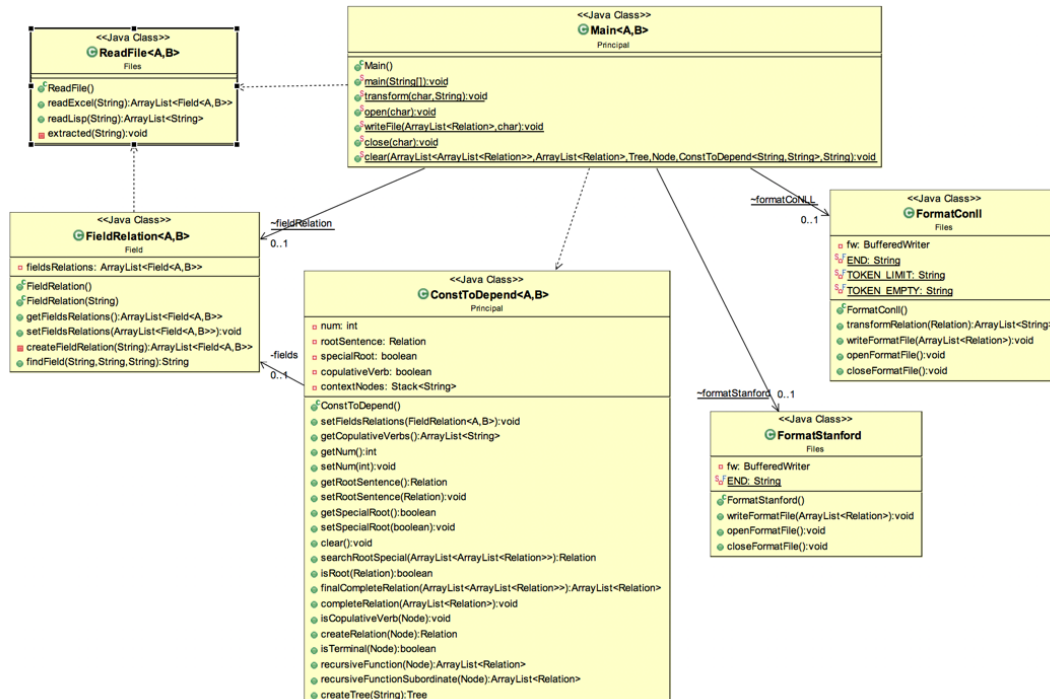


Figura 23. Clase principal de ejecución del algoritmo

El primer paso que se realiza siempre en la ejecución del algoritmo es la carga del fichero que contiene todas las etiquetas asociadas a pares de elementos dentro de un contexto concreto, es decir, contiene las etiquetas que indicarán la funcionalidad de cada una de las relaciones en el modelo de gramática de dependencias.

Esta parte se realiza una única vez, al principio del algoritmo, pues las relaciones no van a ir variando a lo largo del proceso de transformación, sólo pueden cambiar si se modifica el contenido del fichero, caso con el que no habrá inconvenientes, pues este se recarga cada vez que se ejecuta el programa.

La **Figura 24** muestra las clases que intervienen en la lectura de las etiquetas de las relaciones, así como los métodos asociados a ellas y que tienen uso en esta parte del código.

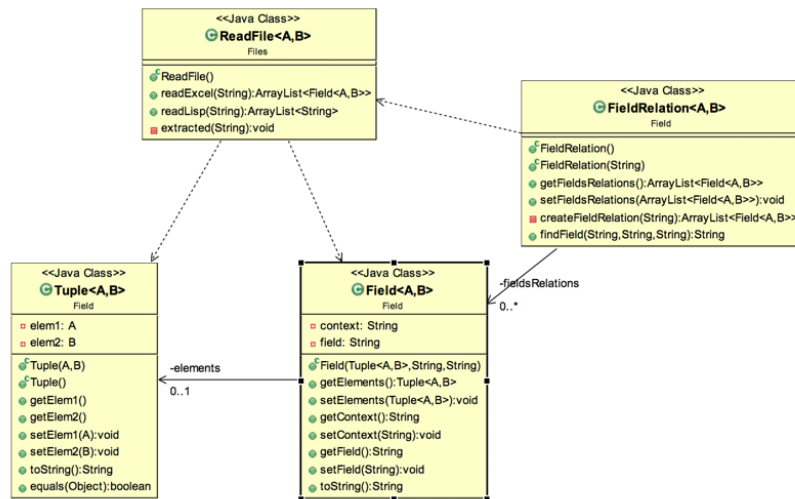


Figura 24. Diagrama de clases de la lectura del fichero de etiquetas

La clase principal en esta parte es *FieldRelation* $\langle A,B \rangle$, una clase con tipos de datos genéricos que contiene los métodos que permiten la creación de la biblioteca de etiquetas, representada por una lista de tipo *Filed* $\langle A,B \rangle$, y la búsqueda de la etiqueta asociada a los dos elementos, es decir, categorías de las palabras que componen una oración, que crean la relación de dependencia dentro de un contexto. Como se explicó en la sección *Transformación de árboles de constituyentes*, el contexto es el constituyente padre de un subárbol que forma parte de la oración.

La composición de la clase *Field* $\langle A,B \rangle$, es una tupla de elementos, es decir, las categorías comentadas previamente además, del contexto y la etiqueta que indica la funcionalidad de dependencia. Por último, tenemos *Tuple* $\langle A,B \rangle$ creada para representar los elementos como tuplas, asemejándose en estructura al formato utilizado por *Stanford Dependencies*, debido a que es más sencillo mantener este formato que una tabla por oración como es el caso del formato CoNLL, en el que recorrer la tabla para actualizar valores implica mayor coste. Se puede observar que la tupla ha sido creada para tipos genéricos, lo que ha implicado que las clases superiores que hacen uso de ella también tengan que tener esa restricción de tipos.

Destacar dentro de *Field* el método ***createFieldRelation(String): ArrayList<Field<A,B>>***, la instanciación de la clase de lectura, en la que se encuentran los métodos de lectura de los diferentes tipos de archivos de entrada que se usan, métodos que se comentarán un poco más adelante. La función del método es la leer el fichero y según se va leyendo línea a línea se van creando las tuplas y etiquetas correspondientes que formarán la biblioteca de éstas.

Una vez que se tiene el listado de etiquetas se pasa a procesar los árboles de constituyentes, proceso que se lleva a cabo en el método ***transform(char, String): void***.

Lo primero es realizar la lectura del fichero *lisp* que contiene los árboles, para ellos se hace uso de la clase *Read* $\langle A,B \rangle$, que contiene un método equivalente al usado para la lectura del fichero de relaciones, este método es ***readLisp(String): ArrayList<String>***, lee el fichero *lisp* línea a línea y compone todas la líneas que pertenecen a la misma oración en una única cadena, guardando esta cadena en una lista que contendrá todas las oraciones del fichero leído.

Es necesario que las oraciones en forma de árbol en el fichero terminen siendo un elemento de la clase *String*, pues el siguiente método utilizado creará una estructura de árbol (*clases Tree y Node*) a partir de una cadena de texto, método reutilizado del trabajo de Borja Colmenajero. El método ***createTree***, recibe una cadena de texto del árbol de constituyentes en representado mediante corchetes o paréntesis y lo transforma en un árbol (*Tree*) con nodos (*Node*).

```

[S
  [NPSUBJ
    [ART "La/ART"]
    [N "policía/N"]]
  [VPTENSED
    [V "descubre/V"]
    [NPOBJ1
      [ART "un/ART"]
      [ADJP
        [ADJ "gran/ADJ"]]
      [N "arsenal/N"]]

```

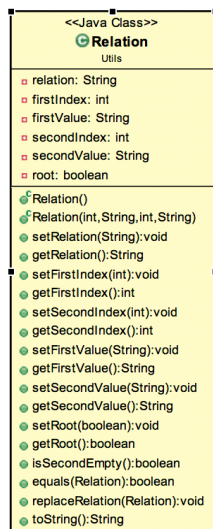
Figura 25. Representación mediante corchetes de un árbol de constituyentes

Esta representación de la oración a transformar se subdivide normalmente en dos subárboles, formados por el sujeto y predica, aunque existe un tercero que está compuesto únicamente por el signo de puntuación final de la oración, como se puede ver en el ejemplo de la **Figura 12**.

Para cada uno de esos subárboles se llama al método ***recursiveFunction(Node): ArrayList<Relation>***, mediante la cual se empezará a recorrer cada uno de ellos, por lo que se tendrá una doble lista de relaciones, es decir, una lista por cada nodo que hereda del constituyente principal *S*, las cuales se incluirán a su vez en otra, teniendo así una especie de matriz de relaciones.

Cuando se empieza a recorrer cada uno de los subárboles se hace uso de varios métodos incluidos en las clases *ConstToDepend* y *Relation*.

Primero veamos un poco la estructura de la clase *Relation*, clase que sirve para la creación de relaciones.



La forma de representar las relaciones está basada en el formato de dependencias de Stanford:

nombre_etiqueta (principal - posición, dependiente - posición)

De forma equivalente con las variables de *Relation*:

relation (firstValue–firstIndex, secondValue–secondIndex)

Además de incorporar otra variable *root* que indica si el elemento *first* (*firstValue* – *firstIndex*) es un nodo raíz dentro del árbol, que se usará posteriormente para completar las relaciones.

Figura 26. Clase *Relation*

Una vez visto como se crean las relaciones pasamos a ver los métodos más importantes y que realizan la transformación de los árboles que se encuentran en la clase *ConstToDepend*, la cual tiene la siguiente estructura.

Como se puede ver se hace uso de las clases explicadas anteriormente *FieldRelation* y *Relation*, pues sin ellas no sería posible crear todas las relaciones que implican la transformación. Además, se utilizan las clases utilizadas del trabajo de Borja Colmenarejo, *Tree* y *Node*, que permitirán recorrer los árboles de constituyentes, haciendo el recorrido en profundidad por los nodos que lo componen.

También se aprecia la existencia de una relación con una clase enumerada llamada *CopulativeVerbs*, que contiene todas las formas verbales existentes en el treebank para los verbos copulativos ser, estar y parecer. Esta clase enumerada es necesaria para identificar si la oración es copulativa y si se debe buscar un “root” de categoría no verbal.

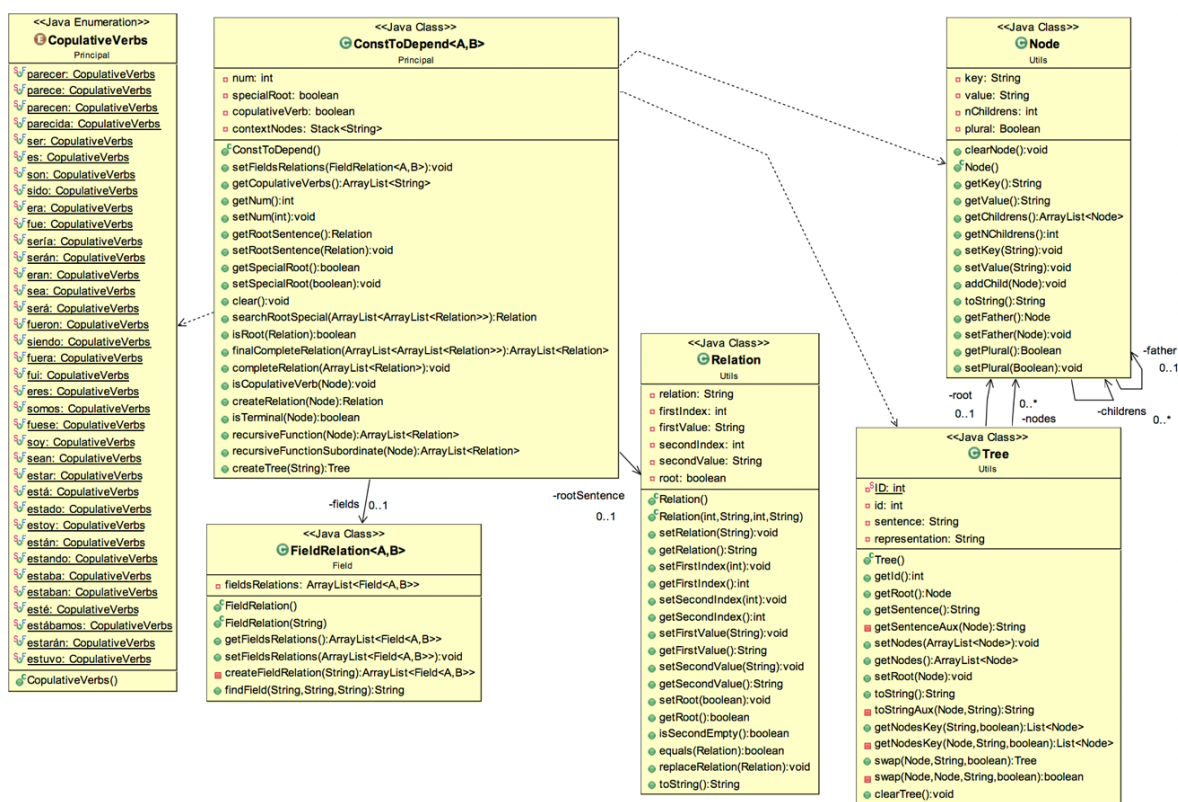


Figura 27. Diagrama de clases enfocado en la clase *ConstToDepend*

En la transformación de cada uno de los árboles leídos en la clase principal *Main*, se utiliza un método recursivo creado en la clase *ConstToDepend*, que tiene como finalidad recorrer el árbol en profundidad hasta llegar a algún nodo terminal, en ese caso se pasa a crear la relación, mientras que en caso contrario se continúa recorriendo el árbol mediante la llamada al mismo método desde el nodo no terminal en el que se encuentra.

```
public ArrayList<Relation> recursiveFunction(Node node, String context) {
    ArrayList<Relation> relations = new ArrayList<>();

    if (isTerminal(node)) { // se comprueba si es terminal
        if (!node.getKey().contains("NPsubj") && !node.getKey().contains("NPattr")) { // se descartan arboles vacios
            num++;
            relations.add(createRelation(node));
        }
    } else { // si no es terminal se guarda el contexto
        context_nodes.push(node.getKey());
    }

    for (Node child : node.getChildren()) { // se recorren los hijos del nodo actual

        if (isTerminal(child)) {
            if (!child.getKey().contains("NPsubj") && !child.getKey().contains("NPattr")) {
                num++;
                relations.add(createRelation(child)); // se crea la relacion asociada al nodo terminal
            }
        } else {
            if (child.getKey().startsWith("CL_")) { // se comprueba si el nodo pertenece a una cláusula subordinada
                relations.addAll(recursiveFunctionSubordinate(child)); // se llama al metodo recursivo que recorrera la subordinada
                                // desde el nodo hijo que lo contiene
            } else {
                relations.addAll(recursiveFunction(child, context)); // se llama al mismo metodo desde el nodo hijo no terminal
            }
        }
    }

    completeRelation(relations); // se completan las relaciones del arbol actual

    if (!context_nodes.isEmpty()) { // eliminar el contexto de la pila
        context_nodes.pop();
    }

    return relations;
}
```

Figura 28. Código del método *recursiveFunction(Node): ArrayList<Relation>*

Este método devuelve una lista con todas las relaciones creadas dentro de un árbol, por lo que después de llamar al método asociado a esa acción, la relación resultante se añade a la lista. De esta forma, a medida que se pasa por un subárbol del árbol principal que forma la oración, se obtiene un pequeño listado de relaciones que se van acumulando en el árbol estrictamente superior al que pertenezcan, completando de forma recursiva la frase, es decir, creando todas las relaciones posibles para cada una de las palabras que la componen.

Hay una particularidad en los casos que implican cláusulas subordinadas, pues se llama a otro método recursivo, que realiza comprobaciones similares dentro de una cláusula subordinada.

Cada vez que se pasa por un nodo no terminal se almacena en la pila de contextos el mismo, el contexto consiste en tomar el nombre del nodo padre del árbol a recorrer, como se comentó anteriormente.

Una vez que se ha llegado al final del árbol y habiendo creado ya toda la relación existente dentro de éste se pasa a completar las relaciones, esto quiere decir que las relaciones hasta el momento solo tienen el primer término asociado, por lo que hay que completar la segunda parte con el nodo “root” del árbol.

Dentro del método *recursiveFunction(Node): ArrayList<Relation>* se aprecia la llamada a otros dos métodos de los que se explicará su funcionamiento, uno de ellos es *createRelation(Node):Relation* y el otro *completeRelation(ArrayList<Relation>): void*

```

public Relation createRelation(Node node) {
    Relation relation = new Relation(num, node.getValue(), 0, "");

    if ((node.getFather().getKey().equals("ADJP_ATTR") && node.getKey().contains("ADJ")) // comprobar si es root por oracion copulativa
        || (node.getFather().getKey().equals("NPATTR")
            && (node.getKey().equals("N") || node.getKey().equals("NP") || node.getKey().equals("NPR"))))
        && copulative_verb) {

        relation.setRoot(true);

        if (root_sentence == null) { // verbo root de la oracion
            // se completa la relacion indicando que esta
            // se establece con un elemento que indica
            // que es el nodo principal de la oracion
            relation.setSecondIndex(0);
            relation.setSecondValue("ROOT");
            relation.setRelation("root");
            root_sentence = new Relation(relation.getFirstIndex(), relation.getFirstValue(),
                relation.getSecondIndex(), relation.getSecondValue());
            root_sentence.setRelation("root");
        }

    } else if (node.getKey().equals("N") || node.getKey().equals("NP") || node.getKey().equals("NPR")) {
        relation.setRoot(true);
    } else if (node.getKey().equals("DATE") && node.getFather().getKey().equals("DATE")) {
        relation.setRoot(true);
    } else if (node.getKey().equals("V") && (
        (node.getFather().getKey().contains("VPTENSED") ) // verbo simple
        || (node.getFather().getKey().contains("VPUNTENSED") ) // clausulas subordinadas: cl_participle, cl_infinitive, ...
        || node.getFather().getKey().equals("V") // verbo compuesto "ha hecho"
        ) ) {

        isCopulativeVerb(node); // comprobar si el verbo es copulativo

        if (!copulative_verb) {
            relation.setRoot(true);
        }

        if (root_sentence == null && context_nodes.contains("VPTENSED") && !copulative_verb) { // verbo root de la oracion
            // se completa la relacion indicando que esta
            // se establece con un elemento que indica
            // que es el nodo principal de la oracion
            relation.setSecondIndex(0);
            relation.setSecondValue("ROOT");
            relation.setRelation("root");
            root_sentence = new Relation(relation.getFirstIndex(), relation.getFirstValue(),
                relation.getSecondIndex(), relation.getSecondValue());
            root_sentence.setRelation("root");
        } else if (root_sentence == null && copulative_verb) {
            relation.setRoot(false);
        }

    } else {
        relation.setRoot(false);
    }

    return relation;
}

```

Figura 29. Código del método *createRelation(Node): Relation*

De este primer método se obtiene la relación incompleta para un nodo terminal dado, se recibe este como argumento de la función y sobre él se realizan diferentes comprobaciones:

- Si la categoría del nodo es un adjetivo o un sustantivo que complementa a los verbos copulativos, en caso de haber encontrado antes uno de ellos, en este caso este nodo es “root”. El flag *copulativeVerb* indica si el verbo encontrado puede ser copulativo o no.
- Se comprueban diferentes casos en los que ciertas categorías hacen que sean “root” del árbol al que pertenecen, por ejemplo, en un sintagma nominal es el sustantivo el “root” de éste.
- Si nos encontramos en un nodo que implica un verbo, es indicador de que estamos en el predicado, por lo que se pasa a crear una relación del verbo como “root” de la frase, exceptuando que sea copulativo, caso en el que no tendrá la propiedad de nodo raíz, haciendo que sea una relación normal.

Una vez procesado el nodo se devuelve la relación que se ha podido crear de éste, si es nodo raíz de la oración, es decir, el verbo principal la relación está completa de la siguiente forma:

root (verbo - posición_verbo, ROOT - 0)

El segundo elemento de la relación en realidad no existe en la frase, solo hace de indicador de que este término es nodo padre del árbol y del que dependerán los demás.

```

public void completeRelation(ArrayList<Relation> relations_node) {
    Relation root_node = null;

    if (!relations_node.parallelStream().filter(p -> p.getRoot()).collect(Collectors.toList()).isEmpty()) {

        root_node = relations_node.parallelStream().
            filter(p -> p.getRoot()).collect(Collectors.toList()).get(0); // obtener el nodo "root" de las relaciones obtenidas

        for (Relation relation : relations_node) { // recorrer la lista de relaciones completandolas
            if (relation.isSecondEmpty() && !relation.equals(root_node)) { // un nodo "root" no se completa consigo mismo
                relation.setSecondIndex(root_node.getFirstIndex());
                relation.setSecondValue(root_node.getFirstValue());
                relation.setRoot(false);

                String field = fields.findField(relation.getFirstValue(), relation.getSecondValue()
                    , context_nodes.lastElement()); // obtener la etiqueta para
                                                    // par de elementos que forman la
                                                    // relacion en el contexto actual

                if (field!=null) {
                    relation.setRelation(field);
                }
            }
        }
    }
}

```

Figura 30. Código del método *completeRelation(ArrayList<Relation>): void*

Ahora pasamos ver como se completan las relaciones dentro de un árbol mediante ***completeRelation(ArrayList<Relation>): void***. Se obtiene el “root” definido previamente, si lo ha habido, dentro del árbol/subárbol. Una vez que se tiene se recorre el listado con todas las relaciones existentes hasta el momento y se asigna en su segundo elemento el nodo padre, exceptuando el propio, pues este dependerá de otro nodo raíz del nivel superior. Cuando se tienen los dos elementos se puede buscar la etiqueta que le corresponde a ese par dependiendo del contexto, por lo que es necesario obtener el último que se ha introducido en la pila, con estos tres elementos se busca la etiqueta y se le asigna, actualizando el valor de la relación en el propio listado de relaciones ya existente.

Siguiendo el código del método ***transform*** de la clase *Main*, una vez que se ha terminado con la llamada a la función recursiva se pasa a otro método ***finalCompleteRelation(ArrayList<ArrayList<Relation>>): ArrayList<Relation>***.

Este método recibe una lista doble de elementos de relaciones, una por cada uno de los árboles principales existentes, el sujeto *NPSUBJ*, si se tiene, y *VPTENED*. Cada uno de ellos tiene asociado un listado con todas las relaciones existentes en sus nodos inferiores. Lo ocurre aquí es que se completa en caso de tener sujeto el nodo padre que se ha obtenido de ese árbol con el nodo raíz de toda la oración, y en el caso del predicado ocurre lo mismo, pues los árboles de dependencias pueden tener más de un elemento dependiendo del “root” principal de la oración. A medida que se completan las relaciones se compone una lista unidimensional con las relaciones.

Ahora está el árbol completo con todas sus relaciones creadas y conectadas dos a dos, por lo que se puede pasar a escribir las relaciones en el documento de salida correspondiente, que como se comentó antes, el tipo de documento que se genera viene dado por el flag *output*.

Todo este proceso explicado para una oración se repite hasta que se ha completado el treebank de constituyentes que se tenía como fuente de entrada, dando lugar a un treebank de dependencias en formato Stanford, CoNLL o ambos. Pero, antes de pasar a la siguiente frase es necesario resetear todos los elementos que se han utilizado en el procesamiento de la oración, para que así no queden restos que puedan afectar a la transformación de la siguiente oración, esto se lleva a cabo mediante el método ***clear()***, que elimina las dependencias que existen de la frase actual, vacía las estructuras *ArrayList* y el árbol que se ha creado a partir de la oración en cadena de texto.

5 Pruebas

A continuación, se hablará de las pruebas realizadas con el código implementado.

5.1 Pruebas de caja negra

Para estas pruebas se hace uso del *plugin* de Eclipse, *JUnit*, con las cuales se probarán algunas clases, aquellas en las que los métodos implementados son más sencillos. Las clases más complicadas tendrán su validación en el apartado **Pruebas de integración**.

CLASE	MÉTODO	DESCRIPCIÓN
RelationTest	equals	Comprueba los elementos de dos relaciones para ver si son iguales
RelationTest	isSecondEmpty	Comprueba que el segundo elemento de la relación se encuentra vacío.
RelationTest	replace	Comprueba que se reemplazan los valores de los elementos de una relación correctamente.
TupleTest	equals	Comprueba los elementos de dos tuplas para ver si son iguales
FieldTest	fieldTest	Comprueba el funcionamiento general de la clase.
FielsRealtionTest	createRelation	Comprueba que se obtiene un listado de etiquetas a partir del fichero.
FielsRealtionTest	findField	Comprueba si se encuentra una etiqueta asociada a dos elementos en un contexto determinado.
FielsRealtionTest	getFieldRelation	Obtiene el listado de etiquetas que se usará posteriormente en la clase <i>ReadFileTest</i> .
ReadTest	readLisp	Comprueba que la lectura de un fichero con extensión lisp, que contiene un treebank de constituyentes, da lugar a un listado de oraciones.
ReadTest	readExcel	Comprueba que la lectura de un fichero Excel, con las relaciones y etiquetas asociadas a estas, da lugar a un listado de etiquetas. Se hace uso de la clase <i>FieldRelationTest</i> .

Tabla 2. Tabla resumen de las pruebas de caja negra

5.2 Pruebas de integración

Dentro de las pruebas de integración se encuentra el funcionamiento principal del código implementado para ellos se ejecuta la clase principal *Main*, con la siguiente línea de ejecución:

-r 'fichero_relaciones' -t 'treebank_constituyentes' -formato_salida

El fichero de relaciones consiste en un archivo Excel, el treebank de constituyentes en documento de extensión lisp y por último el formato de salida del treebank de dependencias, el cual puede ser Stanford (*opción -s*), CoNLL (*opción -c*) o ambos formatos (*opción -b*).

Debido a que el desarrollo del código ha sido incremental, las pruebas de integración, es decir, la comparación del treebank generado, se ha comprado en cada cambio con la correspondencia del treebank de constituyentes en dependencias, el cual ha sido creado por el departamento de lingüística de la Universidad Autónoma de Madrid.

La clase que contiene todos los métodos encargados de las correspondientes transformaciones es *ConstToDepend*, los métodos encargados de recorrer el árbol de constituyentes, de crear las relaciones de los elementos, de completar estas relaciones asignándoles su nodo “*root*” y asignar las etiquetas correspondientes a las relaciones en un contexto determinado haciendo uso de la clase *FieldRelation*.

El desarrollo empezó con oraciones sencillas que no contenían verbos copulativos, oraciones subordinadas u otros elementos más complejos comentados en apartados anteriores, después se fueron añadiendo oraciones con mayor complejidad, haciendo que los métodos tuvieran que poder tratar todos los casos que se iban incluyendo. Según se trataban los distintos tipos de oraciones iban apareciendo particularidades de algunos constituyentes que era necesario tratar, haciendo que se ampliase el código.

Por cada uno de los cambios implementados era necesario realizar la ejecución del código, generando por cada una de ellas una nueva versión del treebank de dependencias, el cual era necesario comparar con el proporcionado por el departamento de lingüística. En estas comparaciones se encontraban fallos en el tratamiento de algunos constituyentes que podían llegar tener algunos casos especiales, algunos de los cuales no pudieron llegar a solventarse como se explica en el apartado **Limitaciones**. Aunque también se pudieron detectar errores en algunos árboles de constituyentes, errores que se comunicaron a los lingüistas.

Cada una de las modificaciones que se realizaban en los métodos que lo requería, ya fuese para añadir funcionalidad o solventar fallos, implicaba volver a revisar todas las oraciones, pues el nuevo código implementado en algunas ocasiones modificaba lo realizado anteriormente, dando lugar a errores que antes no aparecían, por lo que ha sido necesaria una revisión constante y exhaustiva de los resultados, con cada una de las novedades o correcciones introducidas.

Para realizar las comprobaciones sobre el treebank de dependencias generado en cada una de las ejecuciones, se escribía un fichero en formato CoNLL, ya que las oraciones en el modelo de dependencias proporcionadas por los lingüistas se encontraban en este formato. Posteriormente se añadió el formato Stanford, que debía contener la misma información que el generado en formato CoNLL, pero con una representación en forma de pares de elementos con una etiqueta asociada.

6 Calidad de software

Para comprobar la calidad del software realizado se ha hecho uso de dos herramientas una de las cuales permitía su integración en el entorno de desarrollo utilizado, mientras que la otra tiene la opción de usar su versión online.

El entorno utilizado para el desarrollo de código ha sido Eclipse, herramienta utilizada para el desarrollo de software en varios lenguajes, y en este caso utilizado para programar en Java.

Durante el desarrollo se ha seguido una codificación estándar, intentando asegurar en la medida de lo posible, la legibilidad del código siguiendo la guía de estilo publicada por Google, *Google Java Style Code*.⁹

Al finalizar la codificación se ha empleado una herramienta de validación automática denominada **Checkstyle** para asegurar que el código cumple el estándar acordado. El resultado obtenido tras al pasar el validador ha sido satisfactorio, pues ha pasado los test en la mayor parte del código, a excepción de dos casos que se repiten a lo largo de todas las clases implementadas.

El primer error que detecta la herramienta se encuentra en el nombre de los paquetes que conforman el proyecto, pues de acuerdo al estándar estos deben tener una estructura acorde a la siguiente expresión regular: `^[a-z](\.[a-z][a-z0-9]*)*$`. El segundo problema encontrado se da en todos los comentarios Javadoc, tanto en los comentarios de clase como de métodos implementados, según el cual se indica que falta una primera línea en cada uno de los comentarios, y el mensaje que muestra por ello es el siguiente: *Primera frase del Javadoc es incompleta ((período de falta) o no está presente*.

Una vez hecha esta primera comprobación de código se ha decidido utilizar una herramienta de calidad en su versión online, **Kiuwan Code Analysis**. Esta herramienta ha medido cantidad de errores en el código que ha diferenciado por mediante una serie de indicadores y una categorización de la prioridad de los errores hallados.

Los indicadores utilizados son:

- Reliability (**R**): capacidad para mantener un nivel de rendimiento específico.
- Maintainability (**M**): capacidad de ser mejorado incluyendo correcciones, mejora o adaptaciones a cambios de entorno.
- Efficiency (**E**): capacidad para dar un rendimiento adecuado acorde a la cantidad de recursos utilizados.
- Security (**S**): capacidad para proteger información y datos para evitar que personas o sistemas no autorizados puedan acceder a ellos y manipularlos.
- Portability (**P**): capacidad de ser transferido a otro entorno.

⁹ Guía de estilo de Google: <https://google.github.io/styleguide/javaguide.html>

A continuación, se dará un pequeño resumen de los problemas encontrados.

<i>PRIORIDAD</i>	<i>DEFECTOS</i>	<i>INDICADORES MÁS AFECTADOS</i>
Very high	5	(M)
High	51	(E) (R)
Normal	120	(R) (M)
Low	80	(R) (M)
Very low	12	(M)

Tabla 3. Tabla resumen del análisis de la herramienta Kiuwan

La mayoría de los defectos encontrados se encuentran en el ámbito de la mantenibilidad y están referidos a mejorar inicialización de variables e incluso el uso de algunas de ellas, pero lo que más se ve es la recomendación de evitar algunos casos como múltiples comparaciones en estructuras condicionales y nombramiento de métodos o clases. También incluye mejoras a realizar sobre los comentarios de Javadoc.

Para el indicador de confianza (reliability) se encuentran casos similares que, para el punto de mantenibilidad, además, se añaden casos en los que se sería recomendable estructurar más el código incluyendo la mejora de los valores de retorno de algunos métodos.

En los ámbitos de portabilidad y seguridad apenas parecen casos, aunque si destacar dos defectos de nivel alto en la clase encontrada de la lectura de los ficheros de entrada.

Por último, queda la parte de la eficiencia, este indicador se reduce prácticamente en exclusiva a indicar la mejora de varios métodos que contienen bucles, debido a que se emplean varias funciones recursivas que pueden llegar a afectar en este aspecto.

Además, la herramienta ha contabilizado un 100% de código útil en el proyecto que consta de dieciocho ficheros de código implementado, contando las pruebas *JUnit*, que contienen en conjunto una complejidad entorno al 83% y en el que no se ha encontrado código duplicado.

7 Conclusiones y trabajo futuro

7.1 Conclusiones

La transformación del modelo de constituyentes a modelo de dependencias del treebank español de la universidad es un arduo trabajo que requiere la implicación del desarrollador en otras áreas para nada relacionadas con las tecnologías y la informática.

Ha sido necesario recordar numerosos conceptos estudiados previamente relacionadas con la sintaxis de oraciones, además de ampliar tales conocimientos enfocándolos en el correcto funcionamiento de las gramáticas de dependencias. Pero para poder llegar a entenderlas, primero ha sido necesario comprender la de constituyentes. No hay una de las dos que sea más sencilla que la otra, ambas tienen sus particularidades, cosas que posteriormente afectan a la implementación.

En el desarrollo del trabajo se ha conseguido realizar la correcta transformación de muchas estructuras, pero no de todas, debido a las limitaciones comentadas en apartados previos y a que hay gran cantidad de estructuras y no ha sido posible llegar a implementar la correspondencia de todas.

Al igual que ha habido limitaciones con las estructuras, no ha sido posible por parte de los lingüistas llegar a establecer todas las etiquetas que caracterizan las relaciones de dependencias, por lo que no todas tiene asignadas tales.

Concluir, que todo el trabajo realizado ha sido muy gratificante, permitiéndome adquirir numerosos conceptos sobre el procesamiento del lenguaje natural.

7.2 Trabajo futuro

Como trabajo futuro, se puede continuar con la implementación de las reglas de transformación de constituyentes a dependencias, añadiendo un mejor tratamiento de los constituyentes y nueva funcionalidad que solventa las limitaciones que se han ido encontrando y que no han obtenido solución en este proyecto.

La continuación comentada puede llevarse a cabo partiendo del código implementado, sobre el que se pueden aplicar las mejoras e incluir las nuevas funcionalidades.

Otro punto donde sería muy interesante meterse, sería llevar a cabo entrenamientos con modelos de dependencias que están en uso y desarrollo actualmente, como puede ser Stanford Parser para dependencias. De esta forma podría llegar a verse como de bueno es el aprendizaje del modelo para nuestra lengua con las etiquetas establecidas por el departamento de lingüística y no por ellos, de esta forma se podrían realizar comprobaciones con la salida generada por el código implementado, viendo así la proporción de fallos entre ellos.

Referencias

- [1] “Stanford Parser > Neural Network Dependency Parser”, Universidad de Stanford - <https://nlp.stanford.edu/software/nndep.shtml>
- [2] “Stanford Dependencies”, Universidad de Stanford - <https://nlp.stanford.edu/software/stanford-dependencies.shtml>
- [3] “Stanford Parser”, Universidad de Stanford - <https://nlp.stanford.edu/software/lex-parser.shtml>
- [4] “Stanford Parser > Shift-Reduce Constituency Parser”, Universidad de Stanford - <https://nlp.stanford.edu/software/srparser.shtml>
- [5] “Stanford CoreNLP – Natural language software”, Universidad de Stanford - <https://stanfordnlp.github.io/CoreNLP/>
- [6] “Enhanced English Universal Dependencies: An Improved Representation for Natural Language Understanding Tasks by Sebastian Schuster and Christopher D. Manning”, Universidad de Stanford - <https://nlp.stanford.edu/~sebschu/pubs/schuster-manning-lrec2016.pdf>
- [7] “Stanford typed dependencies manual by Marie-Catherine de Marneffe and Christopher D. Manning”, September 2008, Revised for the Stanford Parser v. 3.7.0 in September 2016 - https://nlp.stanford.edu/software/dependencies_manual.pdf
- [8] “Universal Stanford Dependencies: A cross-linguistic typology”, Universidad de Stanford - https://nlp.stanford.edu/pubs/USD_LREC14_paper_camera_ready.pdf
- [9] “Announcing SyntaxNet: The World’s Most Accurate Parser Goes Opens Source “ Google Research Blog - <https://research.googleblog.com/2016/05/announcing-syntaxnet-worlds-most.html>
- [10] “Meet Parsey’s Cousins: Syntax for 40 languages, plus new SyntaxNet capabilities” – Google Research Blog - <https://research.googleblog.com/2016/08/meet-parseys-cousins-syntax-for-40.html>
- [11] “Universal Dependencies” <http://universaldependencies.org>
- [12] “CoNLL-U Format”, Universal Dependencies - <http://universaldependencies.org/format.html>
- [13] “TFG de Borja Colmenarejo García, 2015-2016” - <https://github.com/BorjaC/ValidadorSP/tree/master>
- [14] “A Fundamental Algorithm for Dependency Parsing by Michael A. Covington”, 2001
- [15] “Capítulo de análisis sintáctico by Antonio Moreno Sandoval”

Glosario

Treebank: base de datos compuesta por oraciones en forma de árbol de constituyentes o en formato de dependencias.

Root: forma de referirse a un nodo padre o raíz dentro de un árbol.

Parser: analizador sintáctico.

Cobertura: medida para saber la cantidad de código que se cubre o se usa en las pruebas desarrolladas.

Javadoc: utilidad de Oracle para la generación de documentación de APIs en formato HTML a partir de código fuente java.

JUnit: plugin de Eclipse utilizado para realizar pruebas unitarias en clases java.

CheckStyle: plugin de Eclipse para comprobar la calidad de código frente a un estándar establecido.

EclEmma: plugin que permite medir la cobertura durante las ejecuciones de pruebas.

Plugin: aplicación que se relaciona con otra para agregarle una función nueva y generalmente muy específica.

.jar: tipo de archivo que permite ejecutar aplicaciones escritas en lenguaje java.

Java: lenguaje de programación de propósito general, concurrente y orientado a objetos.

Anexos

A Manual de instalación

La herramienta de transformación está disponible a través del siguiente enlace través del repositorio https://github.com/RebecaPaz/TFG_RebecadelaPaz/tree/master/RebecadelaPaz, pulsar “Clone or download” y a continuación “Download ZIP”. Una vez descargado el archivo comprimido, descomprimir el archivo y se obtiene el proyecto creado en Eclipse, que contiene los ficheros de entrada necesarios para obtener las etiquetas de las relaciones y el treebank de constituyentes utilizado.

Para la correcta ejecución del proyecto, es necesario introducir ciertos parámetros de entrada que indicarán los ficheros a leer y el formato de salida deseado para el treebank de dependencias generado.

La estructura de los parámetros de entrada es la siguiente:

-r ‘fichero_relaciones’ -t ‘treebank_constituyentes’ -formato_salida

La opción **-r** indica el fichero de relaciones que se debe leer. Es necesario que este archivo sea un documento de hoja de cálculo de Excel. Para la lectura de una hoja de cálculo ha sido necesario añadir algunos *.jar* especiales, los se encuentran en la carpeta “jar”, estos son:

commons-codec-1.9.jar
poi-3.11-beta2.jar
poi-ooxml-3.11-beta2.jar
poi-ooxml-schemas-3.11-beta2.jar
xmlbeans-2.6.0.jar

La opción **-t** hace referencia al fichero que contiene el treebank de constituyentes a transformar, este debe ser un documento con extensión *lisp* que contenga los árboles representados mediante una estructura de corchetes, como en la **Figura 11**.

Por último, el formato de salida, las opciones son las siguientes:

-s, formato Stanford
-c, formato CoNLL
-b, ambos

B Manual del programador

La documentación de la herramienta de transformación está disponible a través del siguiente enlace través del repositorio

https://github.com/RebecaPaz/TFG_RebecadelaPaz/tree/master/RebecadelaPaz, pulsar “Clone or download” y a continuación “Download ZIP”. Una vez descargado el archivo comprimido, descomprimir el archivo y obtener la carpeta “doc” que contiene la documentación en javadoc necesario para entender la herramienta.

C Cobertura de las pruebas

Para poder hacer una validación más completa de las pruebas realizadas, se ha utilizado un *plugin* de Eclipse, EclEmma Java Coverage, el cual permite medir la cobertura de un código ejecutado, anotando el porcentaje de uso de los diferentes métodos y clases implicados en la ejecución.

En el proyecto se ha utilizado esta herramienta en las pruebas *JUnit* implementadas para algunas clases y métodos, así como para la ejecución completa del programa a partir de la clase *Main*.

A continuación, se muestran unas tablas resumen de las pruebas llevadas a cabo. Comentar que para estas pruebas no se ha considerado realizar comprobaciones para los métodos *getter* y *setter* existentes en todas las clases, lo que puede llegar a dar un porcentaje bajo de cobertura en alguna de las clases, pues pueden no tener otro tipo de funciones implementadas.

CLASE: ReadTest	
MÉTODO	DESCRIPCIÓN
readLisp	Comprueba que la lectura de un fichero con extensión lisp, que contiene un treebank de constituyentes, da lugar a un listado de oraciones.
readExcel	Comprueba que la lectura de un fichero Excel, con las relaciones y etiquetas asociadas a estas, da lugar a un listado de etiquetas. Se hace uso de la clase <i>FieldRelationTest</i> .
COBERTURA	

Tabla 4. Cobertura de la clase ReadTest

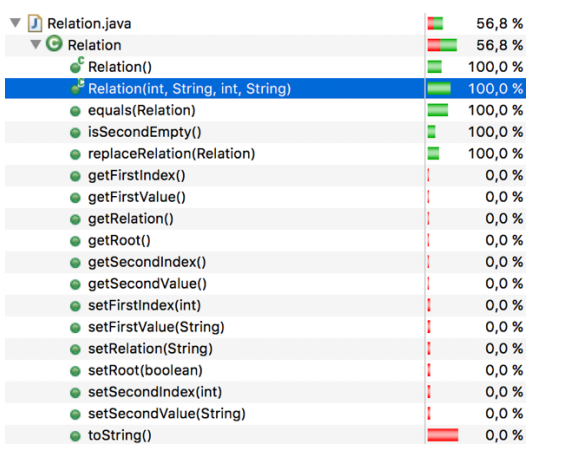
CLASE: RelationTest	
MÉTODO	DESCRIPCIÓN
equals	Comprueba los elementos de dos relaciones para ver si son iguales
isSecondEmpty	Comprueba que el segundo elemento de la relación se encuentra vacío.
replace	Comprueba que se reemplazan los valores de los elementos de una relación correctamente.
COBERTURA	
	

Tabla 5. Cobertura de la clase RelationTest

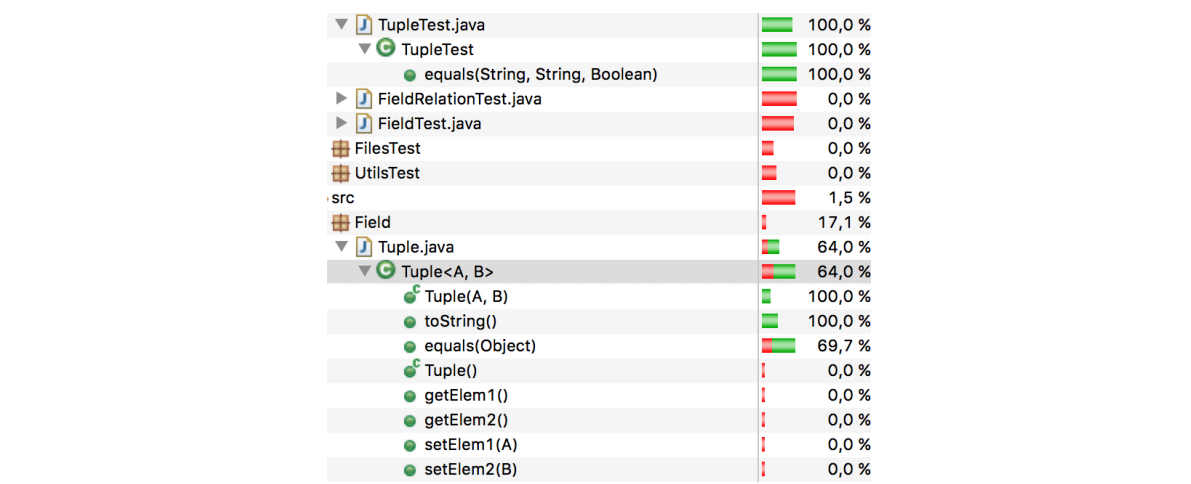
CLASE: TupleTest	
MÉTODO	DESCRIPCIÓN
equals	Comprueba los elementos de dos tuplas para ver si son iguales
COBERTURA	
	

Tabla 6. Cobertura de la clase TupleTest


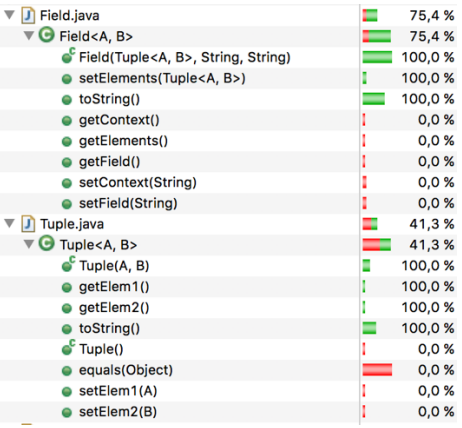
CLASE: FieldTest	
MÉTODO	DESCRIPCIÓN
fieldTest	Comprueba el funcionamiento general de la clase.
COBERTURA	
	

Tabla 7. Cobertura de la clase FieldTest

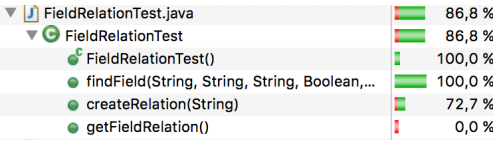
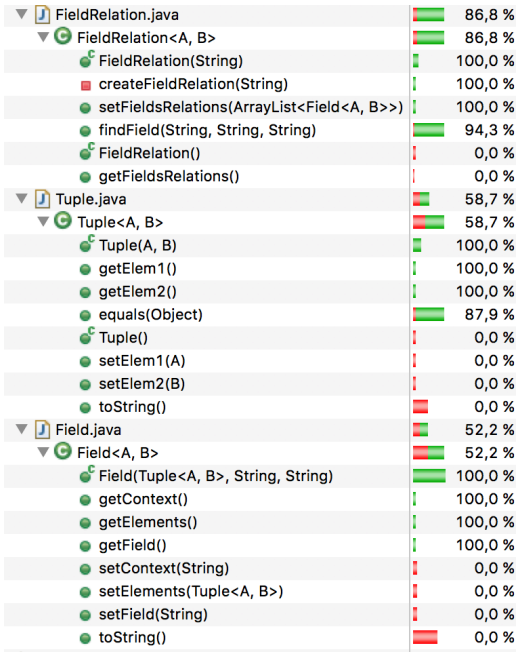
CLASE: FieldRelationTest	
MÉTODO	DESCRIPCIÓN
createRelation	Comprueba que se obtiene un listado de etiquetas a partir del fichero.
findField	Comprueba si se encuentra una etiqueta asociada a dos elementos en un contexto determinado.
getFieldRelation	Obtiene el listado de etiquetas que se usará posteriormente en la clase <i>ReadFileTest</i> .
COBERTURA	
	

Tabla 8. Cobertura de la clase FieldRelationTest

Además, se ha realizado la prueba de cobertura sobre la ejecución de la clase principal, clase *Main*. Debido a la cantidad de clases y métodos de los que hace uso esta ejecución, los resultados de la cobertura se han exportado en formato html.

Los resultados de cobertura de la ejecución principal están disponibles a través del siguiente enlace a través del repositorio https://github.com/RebecaPaz/TFG_RebecadelaPaz/tree/master/RebecadelaPaz, pulsar “Clone or download” y a continuación “Download ZIP”. Una vez descargado el archivo comprimido, descomprimir el archivo y obtener la carpeta “coverage” que contiene los resultados obtenidos de la cobertura.

